

1
2
3
4
5

The OpenACC[®] Application Programming Interface

Version 3.0

OpenACC-Standard.org

November, 2019

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 © 2011-2019 OpenACC-Standard.org. All rights reserved.

11 Contents

12	1. Introduction	9
13	1.1. Scope	9
14	1.2. Execution Model	9
15	1.3. Memory Model	11
16	1.4. Language Interoperability	13
17	1.5. Conventions used in this document	13
18	1.6. Organization of this document	14
19	1.7. References	14
20	1.8. Changes from Version 1.0 to 2.0	16
21	1.9. Corrections in the August 2013 document	17
22	1.10. Changes from Version 2.0 to 2.5	17
23	1.11. Changes from Version 2.5 to 2.6	18
24	1.12. Changes from Version 2.6 to 2.7	19
25	1.13. Changes from Version 2.7 to 3.0	20
26	1.14. Topics Deferred For a Future Revision	21
27	2. Directives	23
28	2.1. Directive Format	23
29	2.2. Conditional Compilation	24
30	2.3. Internal Control Variables	24
31	2.3.1. Modifying and Retrieving ICV Values	24
32	2.4. Device-Specific Clauses	25
33	2.5. Compute Constructs	27
34	2.5.1. Parallel Construct	27
35	2.5.2. Kernels Construct	28
36	2.5.3. Serial Construct	30
37	2.5.4. if clause	32
38	2.5.5. self clause	32
39	2.5.6. async clause	32
40	2.5.7. wait clause	32
41	2.5.8. num_gangs clause	32
42	2.5.9. num_workers clause	32
43	2.5.10. vector_length clause	33
44	2.5.11. private clause	33
45	2.5.12. firstprivate clause	33
46	2.5.13. reduction clause	33
47	2.5.14. default clause	34
48	2.6. Data Environment	35
49	2.6.1. Variables with Predetermined Data Attributes	35
50	2.6.2. Variables with Implicitly Determined Data Attributes	35

51	2.6.3. Data Regions and Data Lifetimes	36
52	2.6.4. Data Structures with Pointers	36
53	2.6.5. Data Construct	37
54	2.6.6. Enter Data and Exit Data Directives	38
55	2.6.7. Reference Counters	40
56	2.6.8. Attachment Counter	41
57	2.7. Data Clauses	41
58	2.7.1. Data Specification in Data Clauses	42
59	2.7.2. Data Clause Actions	43
60	2.7.3. deviceptr clause	46
61	2.7.4. present clause	46
62	2.7.5. copy clause	47
63	2.7.6. copyin clause	47
64	2.7.7. copyout clause	48
65	2.7.8. create clause	49
66	2.7.9. no_create clause	49
67	2.7.10. delete clause	50
68	2.7.11. attach clause	50
69	2.7.12. detach clause	51
70	2.8. Host_Data Construct	51
71	2.8.1. use_device clause	52
72	2.8.2. if clause	52
73	2.8.3. if_present clause	52
74	2.9. Loop Construct	52
75	2.9.1. collapse clause	53
76	2.9.2. gang clause	54
77	2.9.3. worker clause	54
78	2.9.4. vector clause	55
79	2.9.5. seq clause	55
80	2.9.6. auto clause	55
81	2.9.7. tile clause	56
82	2.9.8. device_type clause	56
83	2.9.9. independent clause	56
84	2.9.10. private clause	57
85	2.9.11. reduction clause	57
86	2.10. Cache Directive	61
87	2.11. Combined Constructs	62
88	2.12. Atomic Construct	63
89	2.13. Declare Directive	67
90	2.13.1. device_resident clause	69
91	2.13.2. create clause	69
92	2.13.3. link clause	70
93	2.14. Executable Directives	71
94	2.14.1. Init Directive	71
95	2.14.2. Shutdown Directive	72
96	2.14.3. Set Directive	73
97	2.14.4. Update Directive	74
98	2.14.5. Wait Directive	77

99	2.14.6. Enter Data Directive	77
100	2.14.7. Exit Data Directive	77
101	2.15. Procedure Calls in Compute Regions	77
102	2.15.1. Routine Directive	77
103	2.15.2. Global Data Access	80
104	2.16. Asynchronous Behavior	80
105	2.16.1. async clause	80
106	2.16.2. wait clause	81
107	2.16.3. Wait Directive	82
108	2.17. Fortran Optional Arguments	83
109	3. Runtime Library	85
110	3.1. Runtime Library Definitions	85
111	3.2. Runtime Library Routines	86
112	3.2.1. acc_get_num_devices	86
113	3.2.2. acc_set_device_type	87
114	3.2.3. acc_get_device_type	87
115	3.2.4. acc_set_device_num	88
116	3.2.5. acc_get_device_num	88
117	3.2.6. acc_get_property	89
118	3.2.7. acc_init	90
119	3.2.8. acc_shutdown	91
120	3.2.9. acc_async_test	91
121	3.2.10. acc_async_test_device	92
122	3.2.11. acc_async_test_all	92
123	3.2.12. acc_async_test_all_device	93
124	3.2.13. acc_wait	93
125	3.2.14. acc_wait_device	94
126	3.2.15. acc_wait_async	95
127	3.2.16. acc_wait_device_async	95
128	3.2.17. acc_wait_all	96
129	3.2.18. acc_wait_all_device	96
130	3.2.19. acc_wait_all_async	96
131	3.2.20. acc_wait_all_device_async	97
132	3.2.21. acc_get_default_async	97
133	3.2.22. acc_set_default_async	98
134	3.2.23. acc_on_device	98
135	3.2.24. acc_malloc	99
136	3.2.25. acc_free	99
137	3.2.26. acc_copyin	100
138	3.2.27. acc_create	101
139	3.2.28. acc_copyout	102
140	3.2.29. acc_delete	103
141	3.2.30. acc_update_device	104
142	3.2.31. acc_update_self	105
143	3.2.32. acc_map_data	105
144	3.2.33. acc_unmap_data	106
145	3.2.34. acc_deviceptr	106

146	3.2.35. acc_hostptr	107
147	3.2.36. acc_is_present	107
148	3.2.37. acc_memcpy_to_device	107
149	3.2.38. acc_memcpy_from_device	108
150	3.2.39. acc_memcpy_device	108
151	3.2.40. acc_attach	109
152	3.2.41. acc_detach	109
153	3.2.42. acc_memcpy_d2d	110
154	4. Environment Variables	113
155	4.1. ACC_DEVICE_TYPE	113
156	4.2. ACC_DEVICE_NUM	113
157	4.3. ACC_PROFLIB	113
158	5. Profiling Interface	115
159	5.1. Events	115
160	5.1.1. Runtime Initialization and Shutdown	116
161	5.1.2. Device Initialization and Shutdown	116
162	5.1.3. Enter Data and Exit Data	117
163	5.1.4. Data Allocation	117
164	5.1.5. Data Construct	118
165	5.1.6. Update Directive	118
166	5.1.7. Compute Construct	118
167	5.1.8. Enqueue Kernel Launch	119
168	5.1.9. Enqueue Data Update (Upload and Download)	119
169	5.1.10. Wait	120
170	5.2. Callbacks Signature	120
171	5.2.1. First Argument: General Information	121
172	5.2.2. Second Argument: Event-Specific Information	122
173	5.2.3. Third Argument: API-Specific Information	125
174	5.3. Loading the Library	126
175	5.3.1. Library Registration	127
176	5.3.2. Statically-Linked Library Initialization	128
177	5.3.3. Runtime Dynamic Library Loading	128
178	5.3.4. Preloading with LD_PRELOAD	129
179	5.3.5. Application-Controlled Initialization	130
180	5.4. Registering Event Callbacks	130
181	5.4.1. Event Registration and Unregistration	131
182	5.4.2. Disabling and Enabling Callbacks	132
183	5.5. Advanced Topics	133
184	5.5.1. Dynamic Behavior	134
185	5.5.2. OpenACC Events During Event Processing	135
186	5.5.3. Multiple Host Threads	135
187	6. Glossary	137

188	A. Recommendations for Implementors	141
189	A.1. Target Devices	141
190	A.1.1. NVIDIA GPU Targets	141
191	A.1.2. AMD GPU Targets	141
192	A.1.3. Multicore Host CPU Target	142
193	A.2. API Routines for Target Platforms	142
194	A.2.1. NVIDIA CUDA Platform	143
195	A.2.2. OpenCL Target Platform	144
196	A.3. Recommended Options	145
197	A.3.1. C Pointer in Present clause	145
198	A.3.2. Autoscopying	145
199	Index	147

1. Introduction

This document describes the compiler directives, library routines, and environment variables that collectively define the OpenACC[™] Application Programming Interface (OpenACC API) for writing parallel programs in C, C++, and Fortran that run identified regions in parallel on multicore CPUs or attached accelerators. The method described provides a model for parallel programming that is portable across operating systems and various types of multicore CPUs and accelerators. The directives extend the ISO/ANSI standard C, C++, and Fortran base languages in a way that allows a programmer to migrate applications incrementally to parallel multicore and accelerator targets using standards-based C, C++, or Fortran.

The directives and programming model defined in this document allow programmers to create applications capable of using accelerators without the need to explicitly manage data or program transfers between a host and accelerator or to initiate accelerator startup and shutdown. Rather, these details are implicit in the programming model and are managed by the OpenACC API-enabled compilers and runtime environments. The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops for parallel execution, and similar performance-related details.

1.1. Scope

This OpenACC API document covers only user-directed parallel and accelerator programming, where the user specifies the regions of a program to be targeted for parallel execution. The remainder of the program will be executed sequentially on the host. This document does not describe features or limitations of the host programming environment as a whole; it is limited to specification of loops and regions of code to be executed in parallel on a multicore CPU or an accelerator.

This document does not describe automatic detection of parallel regions or automatic offloading of regions of code to an accelerator by a compiler or other tool. This document does not describe splitting loops or code regions across multiple accelerators attached to a single host. While future compilers may allow for automatic parallelization or automatic offloading, or parallelizing across multiple accelerators of the same type, or across multiple accelerators of different types, these possibilities are not addressed in this document.

1.2. Execution Model

The execution model targeted by OpenACC API-enabled implementations is host-directed execution with an attached parallel accelerator, such as a GPU, or a multicore host with a host thread that initiates parallel execution on the multiple cores, thus treating the multicore CPU itself as a device. Much of a user application executes on a host thread. Compute intensive regions are offloaded to an accelerator or executed on the multiple host cores under control of a host thread. A device, either

234 an attached accelerator or the multicore CPU, executes *parallel regions*, which typically contain
235 work-sharing loops, *kernels regions*, which typically contain one or more loops that may be exe-
236 cuted as kernels, or *serial regions*, which are blocks of sequential code. Even in accelerator-targeted
237 regions, the host thread may orchestrate the execution by allocating memory on the accelerator de-
238 vice, initiating data transfer, sending the code to the accelerator, passing arguments to the compute
239 region, queuing the accelerator code, waiting for completion, transferring results back to the host,
240 and deallocating memory. In most cases, the host can queue a sequence of operations to be executed
241 on a device, one after the other.

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

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

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

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

278 The program starts executing with a single initial host thread, identified by a program counter and

279 its stack. The initial host thread may spawn additional host threads, using OpenACC or another
280 mechanism, such as with the OpenMP API. On a device, a single vector lane of a single worker of a
281 single gang is called a device thread. When executing on an accelerator, a parallel execution context
282 is created on the accelerator and may contain many such threads.

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

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

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

309 1.3. Memory Model

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

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

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

- 324 • Memory bandwidth between host memory and accelerator memory determines the level of
325 compute intensity required to effectively accelerate a given region of code.
- 326 • The user should be aware that a discrete device memory is usually significantly smaller than
327 the host memory, prohibiting offloading regions of code that operate on very large amounts
328 of data.
- 329 • Host addresses stored to pointers on the host may only be valid on the host; addresses stored
330 to pointers in accelerator memory may only be valid on that device. Explicitly transferring
331 pointer values between host and accelerator memory is not advised. Dereferencing host point-
332 ers on an accelerator or dereferencing accelerator pointers on the host is likely to be invalid
333 on such targets.

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

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

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

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

361 1.4. Language Interoperability

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

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

370 1.5. Conventions used in this document

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

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

#pragma acc

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

#pragma acc *directive-name*

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

#pragma acc *directive-name new-line*

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

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

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

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

384 Not all options are allowed in all clauses; the allowable options are clarified for each use of the term
 385 *var*.

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

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

390 1.6. Organization of this document

391 The rest of this document is organized as follows:

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

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

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

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

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

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

404 1.7. References

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

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

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

- 411 – Threads
- 412 – Thread-local storage
- 413 – Parallel memory model
- 414 – Atomic

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

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

- 417 – Thread related features

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

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

- 421 – Extern templates
- 422 – copy and rethrow exceptions
- 423 – memory model
- 424 – atomics
- 425 – move semantics
- 426 – range based loops
- 427 – std::thread
- 428 – thread-local storage
- 429 • ISO/IEC 14882:2014, *Information Technology – Programming Languages – C++*, (C++14).
- 430 • ISO/IEC 14882:2017, *Information Technology – Programming Languages – C++*, (C++17).
- 431 • ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part*
- 432 *1: Base Language*, (Fortran 2003).
- 433 • ISO/IEC 1539-1:2010, *Information Technology – Programming Languages – Fortran – Part*
- 434 *1: Base Language*, (Fortran 2008).
- 435 The use of the following Fortran 2008 features may result in unspecified behavior.
- 436 – Coarrays
- 437 – Do concurrent
- 438 – Simply contiguous arrays rank remapping to rank>1 target
- 439 – Allocatable components of recursive type
- 440 – The block construct
- 441 – Polymorphic assignment
- 442 • ISO/IEC 1539-1:2018, *Information Technology – Programming Languages – Fortran – Part*
- 443 *1: Base Language*, (Fortran 2018).
- 444 The use of the following Fortran 2018 features may result in unspecified behavior.
- 445 – Interoperability with C
- 446 * C functions declared in ISO Fortran binding.h
- 447 * Assumed rank
- 448 – All additional parallel/coarray features
- 449 • *OpenMP Application Program Interface*, version 5.0, November 2018
- 450 • *NVIDIA CUDA[™] C Programming Guide*, version 10.1, May 2019
- 451 • *The OpenCL Specification*, version 2.2, Khronos OpenCL Working Group, July 2019

1.8. Changes from Version 1.0 to 2.0

- `_OPENACC` value updated to 201306
- `default (none)` clause on `parallel` and `kernels` directives
- the implicit data attribute for scalars in `parallel` constructs has changed
- the implicit data attribute for scalars in loops with `loop` directives with the independent attribute has been clarified
- `acc_async_sync` and `acc_async_noval` values for the `async` clause
- Clarified the behavior of the `reduction` clause on a `gang` loop
- Clarified allowable loop nesting (`gang` may not appear inside `worker`, which may not appear within `vector`)
- `wait` clause on `parallel`, `kernels` and `update` directives
- `async` clause on the `wait` directive
- `enter data` and `exit data` directives
- Fortran *common block* names may now appear in many data clauses
- `link` clause for the `declare` directive
- the behavior of the `declare` directive for global data
- the behavior of a data clause with a C or C++ pointer variable has been clarified
- predefined data attributes
- support for multidimensional dynamic C/C++ arrays
- `tile` and `auto` loop clauses
- `update self` introduced as a preferred synonym for `update host`
- `routine` directive and support for separate compilation
- `device_type` clause and support for multiple device types
- nested parallelism using `parallel` or `kernels` region containing another `parallel` or `kernels` region
- `atomic` constructs
- new concepts: `gang-redundant`, `gang-partitioned`; `worker-single`, `worker-partitioned`; `vector-single`, `vector-partitioned`; `thread`
- new API routines:
 - `acc_wait`, `acc_wait_all` instead of `acc_async_wait` and `acc_async_wait_all`
 - `acc_wait_async`
 - `acc_copyin`, `acc_present_or_copyin`
 - `acc_create`, `acc_present_or_create`
 - `acc_copyout`, `acc_delete`

- 486 - **acc_map_data**, **acc_unmap_data**
- 487 - **acc_deviceptr**, **acc_hostptr**
- 488 - **acc_is_present**
- 489 - **acc_memcpy_to_device**, **acc_memcpy_from_device**
- 490 - **acc_update_device**, **acc_update_self**
- 491 • defined behavior with multiple host threads, such as with OpenMP
- 492 • recommendations for specific implementations
- 493 • clarified that no arguments are allowed on the **vector** clause in a parallel region

494 1.9. Corrections in the August 2013 document

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

498 1.10. Changes from Version 2.0 to 2.5

- 499 • The **_OPENACC** value was updated to **201510**; see Section 2.2 Conditional Compilation.
- 500 • The **num_gangs**, **num_workers**, and **vector_length** clauses are now allowed on the
- 501 **kernels** construct; see Section 2.5.2 Kernels Construct.
- 502 • Reduction on C++ class members, array elements, and struct elements are explicitly disal-
- 503 lowed; see Section 2.5.13 reduction clause.
- 504 • Reference counting is now used to manage the correspondence and lifetime of device data;
- 505 see Section 2.6.7 Reference Counters.
- 506 • The behavior of the **exit data** directive has changed to decrement the dynamic reference
- 507 counter. A new optional **finalize** clause was added to set the dynamic reference counter
- 508 to zero. See Section 2.6.6 Enter Data and Exit Data Directives.
- 509 • The **copy**, **copyin**, **copyout**, and **create** data clauses were changed to behave like
- 510 **present_or_copy**, etc. The **present_or_copy**, **pcopy**, **present_or_copyin**,
- 511 **pcopyin**, **present_or_copyout**, **pcopyout**, **present_or_create**, and **pcreate**
- 512 data clauses are no longer needed, though will be accepted for compatibility; see Section 2.7
- 513 Data Clauses.
- 514 • Reductions on orphaned gang loops are explicitly disallowed; see Section 2.9 Loop Construct.
- 515 • The description of the **loop auto** clause has changed; see Section 2.9.6 auto clause.
- 516 • Text was added to the **private** clause on a **loop** construct to clarify that a copy is made
- 517 for each gang or worker or vector lane, not each thread; see Section 2.9.10 private clause.
- 518 • The description of the **reduction** clause on a **loop** construct was corrected; see Sec-
- 519 tion 2.9.11 reduction clause.

- 520 • A restriction was added to the **cache** clause that all references to that variable must lie within
521 the region being cached; see Section 2.10 Cache Directive.
- 522 • Text was added to the **private** and **reduction** clauses on a combined construct to clarify
523 that they act like **private** and **reduction** on the **loop**, not **private** and **reduction**
524 on the **parallel** or **reduction** on the **kernels**; see Section 2.11 Combined Constructs.
- 525 • The **declare create** directive with a Fortran **allocatable** has new behavior; see Sec-
526 tion 2.13.2 create clause.
- 527 • New **init**, **shutdown**, **set** directives were added; see Section 2.14.1 Init Directive, 2.14.2
528 Shutdown Directive, and 2.14.3 Set Directive.
- 529 • A new **if_present** clause was added to the **update** directive, which changes the behavior
530 when data is not present from a runtime error to a no-op; see Section 2.14.4 Update Directive.
- 531 • The **routine bind** clause definition changed; see Section 2.15.1 Routine Directive.
- 532 • An **acc routine** without **gang/worker/vector/seq** is now defined as an error; see
533 Section 2.15.1 Routine Directive.
- 534 • A new **default (present)** clause was added for compute constructs; see Section 2.5.14
535 default clause.
- 536 • The Fortran header file **openacc_lib.h** is no longer supported; the Fortran module **openacc**
537 should be used instead; see Section 3.1 Runtime Library Definitions.
- 538 • New API routines were added to get and set the default async queue value; see Section 3.2.21
539 **acc_get_default_async** and 3.2.22 **acc_set_default_async**.
- 540 • The **acc_copyin**, **acc_create**, **acc_copyout**, and **acc_delete** API routines were
541 changed to behave like **acc_present_or_copyin**, etc. The **acc_present_or_** names
542 are no longer needed, though will be supported for compatibility. See Sections 3.2.26 and fol-
543 lowing.
- 544 • Asynchronous versions of the data API routines were added; see Sections 3.2.26 and follow-
545 ing.
- 546 • A new API routine added, **acc_memcpy_device**, to copy from one device address to
547 another device address; see Section 3.2.37 **acc_memcpy_to_device**.
- 548 • A new OpenACC interface for profile and trace tools was added; see Chapter 5 Profiling Interface.

549 1.11. Changes from Version 2.5 to 2.6

- 550 • The **_OPENACC** value was updated to **201711**.
- 551 • A new **serial** compute construct was added. See Section 2.5.3 Serial Construct.
- 552 • A new runtime API query routine was added. **acc_get_property** may be called from
553 the host and returns properties about any device. See Section 3.2.6.
- 554 • The text has clarified that if a variable is in a reduction which spans two or more nested loops,
555 each **loop** directive on any of those loops must have a **reduction** clause that contains the
556 variable; see Section 2.9.11 reduction clause.

- 557 • An optional **if** or **if_present** clause is now allowed on the **host_data** construct. See
558 Section 2.8 Host_Data Construct.
- 559 • A new **no_create** data clause is now allowed on compute and **data** constructs. See Sec-
560 tion 2.7.9 no_create clause.
- 561 • The behavior of Fortran optional arguments in data clauses and in routine calls has been
562 specified; see Section 2.17 Fortran Optional Arguments.
- 563 • The descriptions of some of the Fortran versions of the runtime library routines were simpli-
564 fied; see Section 3.2 Runtime Library Routines.
- 565 • To allow for manual deep copy of data structures with pointers, new *attach* and *detach* be-
566 havior was added to the data clauses, new **attach** and **detach** clauses were added, and
567 matching **acc_attach** and **acc_detach** runtime API routines were added; see Sections
568 2.6.4, 2.7.11-2.7.12 and 3.2.40-3.2.41.
- 569 • The Intel Coprocessor Offload Interface target and API routine sections were removed from
570 the Section A Recommendations for Implementors, since Intel no longer produces this prod-
571 uct.

572 1.12. Changes from Version 2.6 to 2.7

- 573 • The **_OPENACC** value was updated to **201811**.
- 574 • The specification allows for hosts that share some memory with the device but not all memory.
575 The wording in the text now discusses whether local thread data is in shared memory (memory
576 shared between the local thread and the device) or discrete memory (local thread memory that
577 is not shared with the device), instead of shared-memory devices and non-shared memory
578 devices. See Sections 1.3 Memory Model and 2.6 Data Environment.
- 579 • The text was clarified to allow an implementation that treats a multicore CPU as a device,
580 either an additional device or the only device.
- 581 • The **readonly** modifier was added to the **copyin** data clause and **cache** directive. See
582 Sections 2.7.6 and 2.10.
- 583 • The term *local device* was defined; see Section 1.2 Execution Model and the Glossary.
- 584 • The term *var* is used more consistently throughout the specification to mean a variable name,
585 array name, subarray specification, array element, composite variable member, or Fortran
586 common block name between slashes. Some uses of *var* allow only a subset of these options,
587 and those limitations are given in those cases.
- 588 • The **self** clause was added to the compute constructs; see Section 2.5.5 self clause.
- 589 • The appearance of a **reduction** clause on a compute construct implies a **copy** clause for
590 each reduction variable; see Sections 2.5.13 reduction clause and 2.11 Combined Constructs.
- 591 • The **default (none)** and **default (present)** clauses were added to the **data** con-
592 struct; see Section 2.6.5 Data Construct.
- 593 • Data is defined to be *present* based on the values of the structured and dynamic reference
594 counters; see Section 2.6.7 Reference Counters and the Glossary.

- 595 • The interaction of the **acc_map_data** and **acc_unmap_data** runtime API calls on the
596 present counters is defined; see Section 2.7.2, 3.2.32, and 3.2.33.
- 597 • A restriction clarifying that a **host_data** construct must have at least one **use_device**
598 clause was added.
- 599 • Arrays, subarrays and composite variables are now allowed in **reduction** clauses; see
600 Sections 2.9.11 reduction clause and 2.5.13 reduction clause.
- 601 • Changed behavior of ICVs to support nested compute regions and host as a device semantics.
602 See Section 2.3.

603 1.13. Changes from Version 2.7 to 3.0

- 604 • Updated **_OPENACC** value to **201911**.
- 605 • Updated the normative references to the most recent standards for all base languages. See
606 Section 1.7.
- 607 • Changed the text to clarify uses and limitations of the **device_type** clause and added
608 examples; see Section 2.4.
- 609 • Clarified the conflict between the implicit **copy** clause for variables in a **reduction** clause
610 and the implicit **firstprivate** for scalar variables not in a data clause but used in a
611 **parallel** or **serial** construct; see Sections 2.5.1 and 2.5.3.
- 612 • Required at least one data clause on a **data** construct, an **enter data** directive, or an **exit**
613 **data** directive; see Sections 2.6.5 and 2.6.6.
- 614 • Added text describing how a C++ *lambda* invoked in a compute region and the variables
615 captured by the *lambda* are handled; see Section 2.6.2.
- 616 • Added a **zero** modifier to **create** and **copyout** data clauses that zeros the device memory
617 after it is allocated; see Sections 2.7.7 and 2.7.8.
- 618 • Added a new restriction on the **loop** directive allowing only one of the **seq**, **independent**,
619 and **auto** clauses to appear; see Section 2.9.
- 620 • Added a new restriction on the **loop** directive disallowing a **gang**, **worker**, or **vector**
621 clause to appear if a **seq** clause appears; see Section 2.9.
- 622 • Allowed variables to be modified in an atomic region in a loop where the iterations must
623 otherwise be data independent, such as loops with a **loop independent** clause or a **loop**
624 directive in a **parallel** construct; see Sections 2.9.2, 2.9.3, 2.9.4, and 2.9.9.
- 625 • Clarified the behavior of the **auto** and **independent** clauses on the **loop** directive; see
626 Sections 2.9.6 and 2.9.9.
- 627 • Clarified that an orphaned **loop** construct, or a **loop** construct in a **parallel** construct
628 with no **auto** or **seq** clauses is treated as if an **independent** clause appears; see Sec-
629 tion 2.9.9.
- 630 • For a variable in a **reduction** clause, clarified when the update to the original variable is
631 complete, and added examples; see Section 2.9.11.
- 632 • Clarified that a variable in an orphaned **reduction** clause must be private; see Section 2.9.11.

- 633 • Required at least one clause on a **declare** directive; see Section 2.13.
- 634 • Added an **if** clause to **init**, **shutdown**, **set**, and **wait** directives; see Sections 2.14.1,
635 2.14.2, 2.14.3, and 2.16.3.
- 636 • Required at least one clause on a **set** directive; see Section 2.14.3.
- 637 • Added a *devnum* modifier to the **wait** directive and clause to specify a device to which the
638 wait operation applies; see Section 2.16.3.
- 639 • Allowed a **routine** directive to include a C++ *lambda* name or to appear before a C++
640 *lambda* definition, and defined implicit **routine** directive behavior when a C++ *lambda* is
641 called in a compute region or an *accelerator routine*; see Section 2.15.
- 642 • Added runtime API routine **acc_memcpy_d2d** for copying data directly between two de-
643 vice arrays on the same or different devices; see Section 3.2.42.
- 644 • Defined the values for the **acc_construct_t** and **acc_device_api** enumerations for
645 cross-implementation compatibility; see Sections 5.2.2 and 5.2.3.
- 646 • Changed the return type of **acc_set_cuda_stream** from **int** (values were not specified)
647 to **void**; see Section A.2.1.
- 648 • Edited and expanded Section 1.14 Topics Deferred For a Future Revision.

649 1.14. Topics Deferred For a Future Revision

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

- 655 • Directives to define implicit *deep copy* behavior for pointer-based data structures.
- 656 • Defined behavior when data in data clauses on a directive are aliases of each other.
- 657 • Clarifying when data becomes *present* or *not present* on the device for **enter data** or **exit**
658 **data** directives with an **async** clause.
- 659 • Clarifying the behavior of Fortran **pointer** variables in data clauses.
- 660 • Allowing Fortran **pointer** variables to appear in **deviceptr** clauses.
- 661 • Defining the behavior of data clauses and runtime API routines for pointers that are **NULL**, or
662 Fortran **pointer** variables that are not associated, or Fortran **allocatable** variables that
663 are not allocated.
- 664 • Support for attaching C/C++ pointers that point to an address past the end of a memory region.
- 665 • Fully defined interaction with multiple host threads.
- 666 • Optionally removing the synchronization or barrier at the end of vector and worker loops.
- 667 • Allowing an **if** clause after a **device_type** clause.
- 668 • A **shared** clause (or something similar) for the loop directive.

- 669 • Better support for multiple devices from a single thread, whether of the same type or of
670 different types.
- 671 • An *auto* construct (by some name), to allow **kernels**-like auto-parallelization behavior
672 inside **parallel** constructs or accelerator routines.
- 673 • A **begin declare ... end declare** construct that behaves like putting any global vari-
674 ables declared inside the construct in a **declare** clause.
- 675 • Defining the behavior of parallelism constructs in the base languages when used inside a
676 compute construct or accelerator routine.
- 677 • Optimization directives or clauses, such as an *unroll* directive or clause.
- 678 • Define runtime error behavior and allowing a user-defined error handlers.
- 679 • Extended reductions.
- 680 • Fortran bindings for all the API routines.
- 681 • A **linear** clause for the **loop** directive.
- 682 • Allowing two or more of **gang**, **worker**, **vector**, or **seq** clause on an **acc routine**
683 directive.
- 684 • Requiring the implementation to imply an **acc routine** directive for procedures called
685 within a compute construct or accelerator routine.
- 686 • A single list of all devices of all types, including the host device.
- 687 • A memory allocation API for specific types of memory, including device memory, host pinned
688 memory, and unified memory.
- 689 • A restricted, acceptable form of a loop in a **loop** construct.
- 690 • Bindings to other languages.

2. Directives

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

2.1. Directive Format

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

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

Each directive starts with **#pragma acc**. The remainder of the directive follows the C and C++ conventions for pragmas. 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

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

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

722 Only one *directive-name* can appear per directive, except that a combined directive name is consid-
723 ered a single *directive-name*. The order in which clauses appear is not significant unless otherwise
724 specified. Clauses may be repeated unless otherwise specified. Some clauses have an argument that
725 can contain a list.

726 2.2. Conditional Compilation

727 The `_OPENACC` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is
728 the month designation of the version of the OpenACC directives supported by the implementation.
729 This macro must be defined by a compiler only when OpenACC directives are enabled. The version
730 described here is 201911.

731 2.3. Internal Control Variables

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

736 The ICVs are:

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

741 2.3.1. Modifying and Retrieving ICV Values

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

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

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

752 2.4. Device-Specific Clauses

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

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

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

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

773 A device architecture name may be generic, such as a vendor, or more specific, such as a partic-
774 ular generation of device; see Appendix A Recommendations for Implementors for recommended
775 names. When compiling for a particular device, the implementation will use the clauses associated
776 with the **device_type** clause that specifies the most specific architecture name that applies for
777 this device; clauses associated with any other **device_type** clause are ignored. In this context,

778 the asterisk is the least specific architecture name.

779 **Syntax** The syntax of the `device_type` clause is

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

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

781

782 Examples

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

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

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

```
790 // non-conforming: gang and worker are not permitted together
791 #pragma acc routine gang device_type(foo) worker
```

```
792 // conforming: gang and worker apply to different device types
793 #pragma acc routine device_type(foo) worker \
794 device_type(*) gang
795
```

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

```
799 !$acc parallel num_gangs(2) &
800 !$acc device_type(foo) num_gangs(4) &
801 !$acc device_type(bar) num_workers(8)
```

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

```
804 !$acc parallel device_type(*) num_gangs(2) &
805 !$acc device_type(foo) num_gangs(4) &
806 !$acc device_type(bar) num_workers(8)
```

807

808

809 2.5. Compute Constructs

810 2.5.1. Parallel Construct

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

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

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

813 and in Fortran, the syntax is

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

814 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 )
self [( condition )]
reduction( operator:var-list )
copy( var-list )
copyin( [readonly:]var-list )
copyout( [zero:]var-list )
create( [zero:]var-list )
no_create( var-list )
present( var-list )
deviceptr( var-list )
attach( var-list )
private( var-list )
firstprivate( var-list )
default( none | present )
```

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

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

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

827 If there is no **default (none)** clause on the construct, the compiler will implicitly determine data
 828 attributes for variables that are referenced in the compute construct that do not have predetermined
 829 data attributes and do not appear in a data clause on the compute construct, a lexically containing
 830 **data** construct, or a visible **declare** directive. If there is no **default (present)** clause
 831 on the construct, an array or composite variable referenced in the **parallel** construct that does
 832 not appear in a data clause for the construct or any enclosing **data** construct will be treated as if
 833 it appeared in a **copy** clause for the **parallel** construct. If there is a **default (present)**
 834 clause on the construct, the compiler will implicitly treat all arrays and composite variables without
 835 predetermined data attributes as if they appeared in a **present** clause. A scalar variable referenced
 836 in the **parallel** construct that does not appear in a data clause for the construct or any enclosing
 837 **data** construct will be treated as if it appeared in a **firstprivate** clause unless a reduction
 838 would otherwise imply a **copy** clause for it.

839 Restrictions

- 840 • A program may not branch into or out of an OpenACC **parallel** construct.
- 841 • A program must not depend on the order of evaluation of the clauses, or on any side effects
 842 of the evaluations.
- 843 • Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses
 844 may follow a **device_type** clause.
- 845 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
 846 value; in C or C++, the condition must evaluate to a scalar integer value.
- 847 • At most one **default** clause may appear, and it must have a value of either **none** or
 848 **present**.

849 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
 850 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**
 851 clauses are described in Sections 2.5.11 and Sections 2.5.12. The **device_type** clause is de-
 852 scribed in Section 2.4 Device-Specific Clauses.

853 2.5.2. Kernels Construct

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

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

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

   structured block
```

857 and in Fortran, the syntax is

```

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

```

858 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 )
self [( condition )]
copy( var-list )
copyin( [readonly:]var-list )
copyout( [zero:] var-list )
create( [zero:] var-list )
no_create( var-list )
present( var-list )
deviceptr( var-list )
attach( var-list )
default( none | present )

```

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

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

865 If there is no **default (none)** clause on the construct, the compiler will implicitly determine data
 866 attributes for variables that are referenced in the compute construct that do not have predetermined
 867 data attributes and do not appear in a data clause on the compute construct, a lexically containing
 868 **data** construct, or a visible **declare** directive. If there is no **default (present)** clause
 869 on the construct, an array or composite variable referenced in the **kernels** construct that does
 870 not appear in a data clause for the construct or any enclosing **data** construct will be treated as
 871 if it appeared in a **copy** clause for the **kernels** construct. If there is a **default (present)**
 872 clause on the construct, the compiler will implicitly treat all arrays and composite variables without
 873 predetermined data attributes as if they appeared in a **present** clause. A scalar variable referenced
 874 in the **kernels** construct that does not appear in a data clause for the construct or any enclosing
 875 **data** construct will be treated as if it appeared in a **copy** clause.

876 **Restrictions**

- 877 • A program may not branch into or out of an OpenACC **kernels** construct.
- 878 • A program must not depend on the order of evaluation of the clauses, or on any side effects
879 of the evaluations.
- 880 • Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses
881 may follow a **device_type** clause.
- 882 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
883 value; in C or C++, the condition must evaluate to a scalar integer value.
- 884 • At most one **default** clause may appear, and it must have a value of either **none** or
885 **present**.

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

889 **2.5.3. Serial Construct**

890 **Summary** This construct defines a region of the program that is to be executed sequentially on
891 the current device.

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

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

893 and in Fortran, the syntax is

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

894 where *clause* is one of the following:

```
async [( int-expr )]
wait [( int-expr-list )]
device_type( device-type-list )
if( condition )
self [( condition )]
reduction( operator:var-list )
copy( var-list )
copyin( [readonly:]var-list )
copyout( [zero:] var-list )
create( [zero:] var-list )
no_create( var-list )
```

```

present ( var-list )
deviceptr( var-list )
private( var-list )
firstprivate( var-list )
attach( var-list )
default( none | present )

```

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

900 If the **async** clause does not appear, there is an implicit barrier at the end of the accelerator serial
 901 region, and the execution of the local thread will not proceed until the gang has reached the end of
 902 the serial region.

903 If there is no **default(none)** clause on the construct, the compiler will implicitly determine data
 904 attributes for variables that are referenced in the compute construct that do not have predetermined
 905 data attributes and do not appear in a data clause on the compute construct, a lexically containing
 906 **data** construct, or a visible **declare** directive. If there is no **default(present)** clause
 907 on the construct, an array or composite variable referenced in the **serial** construct that does
 908 not appear in a data clause for the construct or any enclosing **data** construct will be treated as
 909 if it appeared in a **copy** clause for the **serial** construct. If there is a **default(present)**
 910 clause on the construct, the compiler will implicitly treat all arrays and composite variables without
 911 predetermined data attributes as if they appeared in a **present** clause. A scalar variable referenced
 912 in the **serial** construct that does not appear in a data clause for the construct or any enclosing
 913 **data** construct will be treated as if it appeared in a **firstprivate** clause unless a reduction
 914 would otherwise imply a **copy** clause for it.

915 Restrictions

- 916 • A program may not branch into or out of an OpenACC **serial** construct.
- 917 • A program must not depend on the order of evaluation of the clauses, or on any side effects
 918 of the evaluations.
- 919 • Only the **async** and **wait** clauses may follow a **device_type** clause.
- 920 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
 921 value; in C or C++, the condition must evaluate to a scalar integer value.
- 922 • At most one **default** clause may appear, and it must have a value of either **none** or
 923 **present**.

924 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
 925 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**
 926 clauses are described in Sections 2.5.11 and Sections 2.5.12. The **device_type** clause is de-
 927 scribed in Section 2.4 Device-Specific Clauses.

928 2.5.4. **if** clause

929 The **if** clause is optional.

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

933 2.5.5. **self** clause

934 The **self** clause is optional.

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

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

942 2.5.6. **async** clause

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

944 2.5.7. **wait** clause

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

946 2.5.8. **num_gangs** clause

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

953 2.5.9. **num_workers** clause

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

961 2.5.10. **vector_length** clause

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

969 2.5.11. **private** clause

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

972 **Restrictions**

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

975 2.5.12. **firstprivate** clause

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

979 **Restrictions**

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

982 2.5.13. **reduction** clause

983 The **reduction** clause is allowed on the **parallel** and **serial** constructs. It specifies a
984 reduction operator and one or more *vars*. It implies a **copy** data clause for each reduction *var*,
985 unless a data clause for that variable appears on the compute construct. For each reduction *var*, a
986 private copy is created for each parallel gang and initialized for that operator. At the end of the
987 region, the values for each gang are combined using the reduction operator, and the result combined
988 with the value of the original *var* and stored in the original *var*. If the reduction *var* is an array or
989 subarray, the array reduction operation is logically equivalent to applying that reduction operation
990 to each element of the array or subarray individually. If the reduction *var* is a composite variable,
991 the reduction operation is logically equivalent to applying that reduction operation to each member
992 of the composite variable individually. The reduction result is available after the region.

993 The following table lists the operators that are valid and the initialization values; in each case, the
994 initialization value will be cast into the data type of the *var*. For **max** and **min** reductions, the

995 initialization values are the least representable value and the largest representable value for that data
 996 type, respectively. At a minimum, the supported data types include Fortran **logical** as well as
 997 the numerical data types in C (e.g., **_Bool**, **char**, **int**, **float**, **double**, **float _Complex**,
 998 **double _Complex**), C++ (e.g., **bool**, **char**, **wchar_t**, **int**, **float**, **double**), and Fortran
 999 (e.g., **integer**, **real**, **double precision**, **complex**). However, for each reduction operator,
 1000 the supported data types include only the types permitted as operands to the corresponding operator
 1001 in the base language where (1) for max and min, the corresponding operator is less-than and (2) for
 1002 other operators, the operands and the result are the same type.

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

1004 Restrictions

- 1005 • A *var* in a **reduction** clause must be a scalar variable name, a composite variable name,
 1006 an array name, an array element, or a subarray (refer to Section 2.7.1).
- 1007 • If the reduction *var* is an array element or a subarray, accessing the elements of the array
 1008 outside the specified index range results in unspecified behavior.
- 1009 • The reduction *var* may not be a member of a composite variable.
- 1010 • If the reduction *var* is a composite variable, each member of the composite variable must be
 1011 a supported datatype for the reduction operation.
- 1012 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
 1013 **reduction** clauses.

1014 2.5.14. default clause

1015 The **default** clause is optional. The **none** argument tells the compiler to require that all variables
 1016 used in the compute construct that do not have predetermined data attributes to explicitly appear
 1017 in a data clause on the compute construct, a **data** construct that lexically contains the compute
 1018 construct, or a visible **declare** directive. The **present** argument causes all arrays or composite
 1019 variables used in the compute construct that have implicitly determined data attributes to be treated
 1020 as if they appeared in a **present** clause.

2.6. Data Environment

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

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

2.6.1. Variables with Predetermined Data Attributes

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

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

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

2.6.2. Variables with Implicitly Determined Data Attributes

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

1061 appear in a **declare** directive.

1062 **2.6.3. Data Regions and Data Lifetimes**

1063 Data in shared memory is accessible from the current device as well as to the local thread. Such
1064 data is available to the accelerator for the lifetime of the variable. Data not in shared memory must
1065 be copied to and from device memory using data constructs, clauses, and API routines. A *data*
1066 *lifetime* is the duration from when the data is first made available to the accelerator until it becomes
1067 unavailable. For data in shared memory, the data lifetime begins when the data is allocated and
1068 ends when it is deallocated; for statically allocated data, the data lifetime begins when the program
1069 begins and does not end. For data not in shared memory, the data lifetime begins when it is made
1070 present and ends when it is no longer present.

1071 There are four types of data regions. When the program encounters a **data** construct, it creates a
1072 data region.

1073 When the program encounters a compute construct with explicit data clauses or with implicit data
1074 allocation added by the compiler, it creates a data region that has a duration of the compute construct.

1075 When the program enters a procedure, it creates an implicit data region that has a duration of the
1076 procedure. That is, the implicit data region is created when the procedure is called, and exited when
1077 the program returns from that procedure invocation. There is also an implicit data region associated
1078 with the execution of the program itself. The implicit program data region has a duration of the
1079 execution of the program.

1080 In addition to data regions, a program may create and delete data on the accelerator using **enter**
1081 **data** and **exit data** directives or using runtime API routines. When the program executes
1082 an **enter data** directive, or executes a call to a runtime API **acc_copyin** or **acc_create**
1083 routine, each *var* on the directive or the variable on the runtime API argument list will be made live
1084 on accelerator.

1085 **2.6.4. Data Structures with Pointers**

1086 This section describes the behavior of data structures that contain pointers. A pointer may be a
1087 C or C++ pointer (e.g., **float***), a Fortran pointer or array pointer (e.g., **real, pointer,**
1088 **dimension(:)**), or a Fortran allocatable (e.g., **real, allocatable, dimension(:)**).

1089 When a data object is copied to device memory, the values are copied exactly. If the data is a data
1090 structure that includes a pointer, or is just a pointer, the pointer value copied to device memory
1091 will be the host pointer value. If the pointer target object is also allocated in or copied to device
1092 memory, the pointer itself needs to be updated with the device address of the target object before
1093 dereferencing the pointer in device memory.

1094 An *attach* action updates the pointer in device memory to point to the device copy of the data
1095 that the host pointer targets; see Section 2.7.2. For Fortran array pointers and allocatable arrays,
1096 this includes copying any associated descriptor (dope vector) to the device copy of the pointer.
1097 When the device pointer target is deallocated, the pointer in device memory should be restored
1098 to the host value, so it can be safely copied back to host memory. A *detach* action updates the
1099 pointer in device memory to have the same value as the corresponding pointer in local memory;
1100 see Section 2.7.2. The *attach* and *detach* actions are performed by the **copy**, **copyin**, **copyout**,

1101 **create**, **attach**, and **detach** data clauses (Sections 2.7.3-2.7.12), and the **acc_attach** and
 1102 **acc_detach** runtime API routines (Sections 3.2.40 and 3.2.41). The *attach* and *detach* actions
 1103 use attachment counters to determine when the pointer in device memory needs to be updated; see
 1104 Section 2.6.8.

1105 2.6.5. Data Construct

1106 **Summary** The **data** construct defines *vars* to be allocated in the current device memory for
 1107 the duration of the region, whether data should be copied from local memory to the current device
 1108 memory upon region entry, and copied from device memory to local memory upon region exit.

1109 **Syntax** In C and C++, the syntax of the OpenACC **data** construct is

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

1110 and in Fortran, the syntax is

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

1111 where *clause* is one of the following:

```
if( condition )
copy( var-list )
copyin( [readonly:]var-list )
copyout( [zero:]var-list )
create( [zero:]var-list )
no_create( var-list )
present( var-list )
deviceptr( var-list )
attach( var-list )
default( none | present )
```

1112 **Description** Data will be allocated in the memory of the current device and copied from local
 1113 memory to device memory, or copied back, as required. The data clauses are described in Sec-
 1114 tion 2.7 Data Clauses. Structured reference counters are incremented for data when entering a data
 1115 region, and decremented when leaving the region, as described in Section 2.6.7 Reference Counters.

1116 Restrictions

- 1117 • At least one **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**,
 1118 **attach**, or **default** clause must appear on a **data** construct.

1119 **if clause**

1120 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate
 1121 space in the current device memory and move data from and to the local memory as required.
 1122 When an **if** clause appears, the program will conditionally allocate memory in and move data to
 1123 and/or from device memory. When the *condition* in the **if** clause evaluates to zero in C or C++, or
 1124 **.false.** in Fortran, no device memory will be allocated, and no data will be moved. When the
 1125 *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the data will be allocated and
 1126 moved as specified. At most one **if** clause may appear.

1127 **default clause**

1128 The **default** clause is optional. If the **default** clause is present, then for each compute construct
 1129 that is lexically contained within the data construct the behavior will be as if a **default** clause with
 1130 the same value appeared on the compute construct, unless a **default** clause already appears on
 1131 the compute construct. At most one **default** clause may appear.

1132 **2.6.6. Enter Data and Exit Data Directives**

1133 **Summary** An **enter data** directive may be used to define *vars* to be allocated in the current
 1134 device memory for the remaining duration of the program, or until an **exit data** directive that
 1135 deallocates the data. They also tell whether data should be copied from local memory to device
 1136 memory at the **enter data** directive, and copied from device memory to local memory at the
 1137 **exit data** directive. The dynamic range of the program between the **enter data** directive and
 1138 the matching **exit data** directive is the data lifetime for that data.

1139 **Syntax** In C and C++, the syntax of the OpenACC **enter data** directive is

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

1140 and in Fortran, the syntax is

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

1141 where *clause* is one of the following:

```
if( condition )  

async [( int-expr )]  

wait [( wait-argument )]  

copyin( var-list )  

create( [zero:]var-list )  

attach( var-list )
```

1142 In C and C++, the syntax of the OpenACC **exit data** directive is

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

1143 and in Fortran, the syntax is

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

1144 where *clause* is one of the following:

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

1145 **Description** At an **enter data** directive, data may be allocated in the current device mem-
 1146 ory and copied from local memory to device memory. This action enters a data lifetime for those
 1147 *vars*, and will make the data available for **present** clauses on constructs within the data life-
 1148 time. Dynamic reference counters are incremented for this data, as described in Section 2.6.7
 1149 Reference Counters. Pointers in device memory may be *attached* to point to the corresponding
 1150 device copy of the host pointer target.

1151 At an **exit data** directive, data may be copied from device memory to local memory and deal-
 1152 located from device memory. If no **finalize** clause appears, dynamic reference counters are
 1153 decremented for this data. If a **finalize** clause appears, the dynamic reference counters are set
 1154 to zero for this data. Pointers in device memory may be *detached* so as to have the same value as
 1155 the original host pointer.

1156 The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is de-
 1157 scribed in Section 2.6.7 Reference Counters.

1158 Restrictions

- 1159 • At least one **copyin**, **create**, or **attach** clause must appear on an **enter data** direc-
 1160 tive.
- 1161 • At least one **copyout**, **delete**, or **detach** clause must appear on an **exit data** direc-
 1162 tive.

1163 if clause

1164 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or
 1165 deallocate space in the current device memory and move data from and to local memory. When an
 1166 **if** clause appears, the program will conditionally allocate or deallocate device memory and move
 1167 data to and/or from device memory. When the *condition* in the **if** clause evaluates to zero in C or
 1168 C++, or **.false.** in Fortran, no device memory will be allocated or deallocated, and no data will
 1169 be moved. When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the data
 1170 will be allocated or deallocated and moved as specified.

1171 async clause

1172 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1173 wait clause

1174 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1175 finalize clause

1176 The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize**
1177 clause appears, the **exit data** directive will decrement the dynamic reference counters for *vars*
1178 appearing in **copyout** and **delete** clauses, and will decrement the attachment counters for point-
1179 ers appearing in **detach** clauses. If a **finalize** clause appears, the **exit data** directive will
1180 set the dynamic reference counters to zero for *vars* appearing in **copyout** and **delete** clauses,
1181 and will set the attachment counters to zero for pointers appearing in **detach** clauses.

1182 2.6.7. Reference Counters

1183 When device memory is allocated for data not in shared memory due to data clauses or OpenACC
1184 API routine calls, the OpenACC implementation keeps track of that device memory and its relation-
1185 ship to the corresponding data in host memory.

1186 Each section of device memory will be associated with two *reference counters* per device, a struc-
1187 tured reference counter and a dynamic reference counter. The structured and dynamic reference
1188 counters are used to determine when to allocate or deallocate data in device memory. The struc-
1189 tured reference counter for a block of data keeps track of how many nested data regions have been
1190 entered for that data. The initial value of the structured reference counter for static data in device
1191 memory (in a global **declare** directive) is one; for all other data, the initial value is zero. The
1192 dynamic reference counter for a block of data keeps track of how many dynamic data lifetimes are
1193 currently active in device memory for that block. The initial value of the dynamic reference counter
1194 is zero. Data is considered *present* if the sum of the structured and dynamic reference counters is
1195 greater than zero.

1196 A structured reference counter is incremented when entering each data or compute region that con-
1197 tain an explicit data clause or implicitly-determined data attributes for that block of memory, and
1198 is decremented when exiting that region. A dynamic reference counter is incremented for each
1199 **enter data copyin** or **create** clause, or each **acc_copyin** or **acc_create** API routine
1200 call for that block of memory. The dynamic reference counter is decremented for each **exit data**
1201 **copyout** or **delete** clause when no **finalize** clause appears, or each **acc_copyout** or
1202 **acc_delete** API routine call for that block of memory. The dynamic reference counter will be
1203 set to zero with an **exit data copyout** or **delete** clause when a **finalize** clause appears,
1204 or each **acc_copyout_finalize** or **acc_delete_finalize** API routine call for the block
1205 of memory. The reference counters are modified synchronously with the local thread, even if the
1206 data directives include an **async** clause. When both structured and dynamic reference counters
1207 reach zero, the data lifetime in device memory for that data ends.

1208 2.6.8. Attachment Counter

1209 Since multiple pointers can target the same address, each pointer in device memory is associated
1210 with an *attachment counter* per device. The *attachment counter* for a pointer is initialized to zero
1211 when the pointer is allocated in device memory. The *attachment counter* for a pointer is set to one
1212 whenever the pointer is *attached* to new target address, and incremented whenever an *attach* action
1213 for that pointer is performed for the same target address. The *attachment counter* is decremented
1214 whenever a *detach* action occurs for the pointer, and the pointer is *detached* when the *attachment*
1215 *counter* reaches zero. This is described in more detail in Section 2.7.2 Data Clause Actions.

1216 A pointer in device memory can be assigned a device address in two ways. The pointer can be
1217 attached to a device address due to data clauses or API routines, as described in Section 2.7.2
1218 Data Clause Actions, or the pointer can be assigned in a compute region executed on that device.
1219 Unspecified behavior may result if both ways are used for the same pointer.

1220 Pointer members of structs, classes, or derived types in device or host memory can be overwritten
1221 due to update directives or API routines. It is the user's responsibility to ensure that the pointers
1222 have the appropriate values before or after the data movement in either direction. The behavior of
1223 the program is undefined if any of the pointer members are attached when an update of a composite
1224 variable is performed.

1225 2.7. Data Clauses

1226 These data clauses may appear on the **parallel** construct, **kernels** construct, **serial** con-
1227 struct, **data** construct, the **enter data** and **exit data** directives, and **declare** directives.
1228 In the descriptions, the *region* is a compute region with a clause appearing on a **parallel**,
1229 **kernels**, or **serial** construct, a data region with a clause on a **data** construct, or an implicit
1230 data region with a clause on a **declare** directive. If the **declare** directive appears in a global
1231 context, the corresponding implicit data region has a duration of the program. The list argument to
1232 each data clause is a comma-separated collection of *vars*. For all clauses except **deviceptr** and
1233 **present**, the list argument may include a Fortran *common block* name enclosed within slashes,
1234 if that *common block* name also appears in a **declare** directive **link** clause. In all cases, the
1235 compiler will allocate and manage a copy of the *var* in the memory of the current device, creating a
1236 visible device copy of that *var*, for data not in shared memory.

1237 OpenACC supports accelerators with discrete memories from the local thread. However, if the
1238 accelerator can access the local memory directly, the implementation may avoid the memory allo-
1239 cation and data movement and simply share the data in local memory. Therefore, a program that
1240 uses and assigns data on the host and uses and assigns the same data on the accelerator within a
1241 data region without update directives to manage the coherence of the two copies may get different
1242 answers on different accelerators or implementations.

1243 Restrictions

- 1244 • Data clauses may not follow a **device_type** clause.
- 1245 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
1246 data clauses.

1247 2.7.1. Data Specification in Data Clauses

1248 In C and C++, a subarray is an array name followed by an extended array range specification in
1249 brackets, with start and length, such as

AA[2:n]

1250 If the lower bound is missing, zero is used. If the length is missing and the array has known size, the
1251 size of the array is used; otherwise the length is required. The subarray **AA[2:n]** means element
1252 **AA[2], AA[3], ..., AA[2+n-1]**.

1253 In C and C++, a two dimensional array may be declared in at least four ways:

- 1254 • Statically-sized array: **float AA[100][200];**
- 1255 • Pointer to statically sized rows: **typedef float row[200]; row* BB;**
- 1256 • Statically-sized array of pointers: **float* CC[200];**
- 1257 • Pointer to pointers: **float** DD;**

1258 Each dimension may be statically sized, or a pointer to dynamically allocated memory. Each of
1259 these may be included in a data clause using subarray notation to specify a rectangular array:

- 1260 • **AA[2:n][0:200]**
- 1261 • **BB[2:n][0:m]**
- 1262 • **CC[2:n][0:m]**
- 1263 • **DD[2:n][0:m]**

1264 Multidimensional rectangular subarrays in C and C++ may be specified for any array with any com-
1265 bination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions,
1266 all dimensions except the first must specify the whole extent, to preserve the contiguous data re-
1267 striction, discussed below. For dynamically allocated dimensions, the implementation will allocate
1268 pointers in device memory corresponding to the pointers in local memory, and will fill in those
1269 pointers as appropriate.

1270 In Fortran, a subarray is an array name followed by a comma-separated list of range specifications
1271 in parentheses, with lower and upper bound subscripts, such as

arr(1:high, low:100)

1272 If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if
1273 known, are used. All dimensions except the last must specify the whole extent, to preserve the
1274 contiguous data restriction, discussed below.

1275 Restrictions

- 1276 • In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be
1277 specified.

- 1278 • In C and C++, the length for dynamically allocated dimensions of an array must be explicitly
1279 specified.
- 1280 • In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host
1281 or on the device, may result in undefined behavior.
- 1282 • If a subarray appears in a data clause, the implementation may choose to allocate memory for
1283 only that subarray on the accelerator.
- 1284 • In Fortran, array pointers may appear, but pointer association is not preserved in device mem-
1285 ory.
- 1286 • Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous
1287 block of memory, except for dynamic multidimensional C arrays.
- 1288 • In C and C++, if a variable or array of composite type appears, all the data members of the
1289 struct or class are allocated and copied, as appropriate. If a composite member is a pointer
1290 type, the data addressed by that pointer are not implicitly copied.
- 1291 • In Fortran, if a variable or array of composite type appears, all the members of that derived
1292 type are allocated and copied, as appropriate. If any member has the **allocatable** or
1293 **pointer** attribute, the data accessed through that member are not copied.
- 1294 • If an expression is used in a subscript or subarray expression in a clause on a **data** construct,
1295 the same value is used when copying data at the end of the data region, even if the values of
1296 variables in the expression change during the data region.

1297 2.7.2. Data Clause Actions

1298 Most of the data clauses perform one or more the following actions. The actions test or modify one
1299 or both of the structured and dynamic reference counters, depending on the directive on which the
1300 data clause appears.

1301 Present Increment Action

1302 A *present increment* action is one of the actions that may be performed for a **present** (Section
1303 2.7.4), **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Sec-
1304 tion 2.7.8), or **no_create** (Section 2.7.9) clause, or for a call to an **acc_copyin** (Section 3.2.26)
1305 or **acc_create** (Section 3.2.27) API routine. See those sections for details.

1306 A *present increment* action for a *var* occurs only when *var* is already present in device memory.

1307 A *present increment* action for a *var* increments the structured or dynamic reference counter for *var*.

1308 Present Decrement Action

1309 A *present decrement* action is one of the actions that may be performed for a **present** (Section
1310 2.7.4), **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Sec-
1311 tion 2.7.8), **no_create** (Section 2.7.9), or **delete** (Section 2.7.10) clause, or for a call to an
1312 **acc_copyout** (Section 3.2.28) or **acc_delete** (Section 3.2.29) API routine. See those sec-
1313 tions for details.

1314 A *present decrement* action for a *var* occurs only when *var* is already present in device memory.

1315 A *present decrement* action for a *var* decrements the structured or dynamic reference counter for
1316 *var*, if its value is greater than zero. If the device memory associated with *var* was mapped to
1317 the device using **acc_map_data**, the dynamic reference count may not be decremented to zero,
1318 except by a call to **acc_unmap_data**. If the reference counter is already zero, its value is left
1319 unchanged.

1320 **Create Action**

1321 A *create* action is one of the actions that may be performed for a **copyout** (Section 2.7.7) or
1322 **create** (Section 2.7.8) clause, or for a call to an **acc_create** API routine (Section 3.2.27). See
1323 those sections for details.

1324 A *create* action for a *var* occurs only when *var* is not already present in device memory.

1325 A *create* action for a *var*:

- 1326 • allocates device memory for *var*; and
- 1327 • sets the structured or dynamic reference counter to one.

1328 **Copyin Action**

1329 A *copyin* action is one of the actions that may be performed for a **copy** (Section 2.7.5) or **copyin**
1330 (Section 2.7.6) clause, or for a call to an **acc_copyin** API routine (Section 3.2.26). See those
1331 sections for details.

1332 A *copyin* action for a *var* occurs only when *var* is not already present in device memory.

1333 A *copyin* action for a *var*:

- 1334 • allocates device memory for *var*;
- 1335 • initiates a copy of the data for *var* from the local thread memory to the corresponding device
1336 memory; and
- 1337 • sets the structured or dynamic reference counter to one.

1338 The data copy may complete asynchronously, depending on other clauses on the directive.

1339 **Copyout Action**

1340 A *copyout* action is one of the actions that may be performed for a **copy** (Section 2.7.5) or
1341 **copyout** (Section 2.7.7) clause, or for a call to an **acc_copyout** API routine (Section 3.2.28).
1342 See those sections for details.

1343 A *copyout* action for a *var* occurs only when *var* is present in device memory.

1344 A *copyout* action for a *var*:

- 1345 • performs an *immediate detach* action for any pointer in *var*;
- 1346 • initiates a copy of the data for *var* from device memory to the corresponding local thread
1347 memory; and

- 1348 • deallocates device memory for *var*.

1349 The data copy may complete asynchronously, depending on other clauses on the directive, in which
1350 case the memory is deallocated when the data copy is complete.

1351 Delete Action

1352 A *delete* action is one of the actions that may be performed for a **present** (Section 2.7.4), **copyin**
1353 (Section 2.7.6), **create** (Section 2.7.8), **no_create** (Section 2.7.9), or **delete** (Section 2.7.10)
1354 clause, or for a call to an **acc_delete** API routine (Section 3.2.29). See those sections for details.

1355 A *delete* action for a *var* occurs only when *var* is present in device memory.

1356 A *delete* action for *var*:

- 1357 • performs an *immediate detach* action for any pointer in *var*; and
1358 • deallocates device memory for *var*.

1359 Attach Action

1360 An *attach* action is one of the actions that may be performed for a **present** (Section 2.7.4),
1361 **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Section 2.7.8),
1362 **no_create** (Section 2.7.9), or **attach** (Section 2.7.10) clause, or for a call to an **acc_attach**
1363 API routine (Section 3.2.40). See those sections for details.

1364 An *attach* action for a *var* occurs only when *var* is a pointer reference.

1365 If the pointer *var* is in shared memory or is not present in the current device memory, or if the
1366 address to which *var* points is not present in the current device memory, no action is taken. If the
1367 *attachment counter* for *var* is nonzero and the pointer in device memory already points to the device
1368 copy of the data in *var*, the *attachment counter* for the pointer *var* is incremented. Otherwise, the
1369 pointer in device memory is *attached* to the device copy of the data by initiating an update for the
1370 pointer in device memory to point to the device copy of the data and setting the *attachment counter*
1371 for the pointer *var* to one. The update may complete asynchronously, depending on other clauses
1372 on the directive. The pointer update must follow any data copies due to *copyin* actions that are
1373 performed for the same directive.

1374 Detach Action

1375 A *detach* action is one of the actions that may be performed for a **present** (Section 2.7.4),
1376 **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Section 2.7.8),
1377 **no_create** (Section 2.7.9), **delete** (Section 2.7.10), or **detach** (Section 2.7.10) clause, or for
1378 a call to an **acc_detach** API routine (Section 3.2.41). See those sections for details.

1379 A *detach* action for a *var* occurs only when *var* is a pointer reference.

1380 If the pointer *var* is in shared memory or is not present in the current device memory, or if the
1381 *attachment counter* for *var* for the pointer is zero, no action is taken. Otherwise, the *attachment*
1382 *counter* for the pointer *var* is decremented. If the *attachment counter* is decreased to zero, the
1383 pointer is *detached* by initiating an update for the pointer *var* in device memory to have the same

1384 value as the corresponding pointer in local memory. The update may complete asynchronously,
 1385 depending on other clauses on the directive. The pointer update must precede any data copies due
 1386 to *copyout* actions that are performed for the same directive.

1387 Immediate Detach Action

1388 An *immediate detach* action is one of the actions that may be performed for a **detach** (Section
 1389 2.7.10) clause, or for a call to an **acc_detach_finalize** API routine (Section 3.2.41). See
 1390 those sections for details.

1391 An *immediate detach* action for a *var* occurs only when *var* is a pointer reference and is present in
 1392 device memory.

1393 If the *attachment counter* for the pointer is zero, the *immediate detach* action has no effect. Other-
 1394 wise, the *attachment counter* for the pointer set to zero and the pointer is *detached* by initiating an
 1395 update for the pointer in device memory to have the same value as the corresponding pointer in local
 1396 memory. The update may complete asynchronously, depending on other clauses on the directive.
 1397 The pointer update must precede any data copies due to *copyout* actions that are performed for the
 1398 same directive.

1399 2.7.3. deviceptr clause

1400 The **deviceptr** clause may appear on structured **data** and compute constructs and **declare**
 1401 directives.

1402 The **deviceptr** clause is used to declare that the pointers in *var-list* are device pointers, so the
 1403 data need not be allocated or moved between the host and device for this pointer.

1404 In C and C++, the *vars* in *var-list* must be pointer variables.

1405 In Fortran, the *vars* in *var-list* must be dummy arguments (arrays or scalars), and may not have the
 1406 Fortran **pointer**, **allocatable**, or **value** attributes.

1407 For data in shared memory, host pointers are the same as device pointers, so this clause has no
 1408 effect.

1409 2.7.4. present clause

1410 The **present** clause may appear on structured **data** and compute constructs and **declare** di-
 1411 rectives. The **present** clause specifies that *vars* in *var-list* are in shared memory or are already
 1412 present in the current device memory due to data regions or data lifetimes that contain the construct
 1413 on which the **present** clause appears.

1414 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1415 the **present** clause behaves as follows:

- 1416 • At entry to the region:
 - 1417 – If *var* is not present in the current device memory, a runtime error is issued.
 - 1418 – Otherwise, a *present increment* action with the structured reference counter is performed.
 - 1419 – If *var* is a pointer reference, an *attach* action is performed.

- 1420 • At exit from the region:
- 1421 – If *var* is not present in the current device memory, a runtime error is issued.
- 1422 – Otherwise, a *present decrement* action with the structured reference counter is per-
- 1423 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
- 1424 and dynamic reference counters are zero, a *delete* action is performed.

1425 Restrictions

- 1426 • If only a subarray of an array is present in the current device memory, the **present** clause
- 1427 must specify the same subarray, or a subarray that is a proper subset of the subarray in the
- 1428 data lifetime.
- 1429 • It is a runtime error if the subarray in *var-list* clause includes array elements that are not part
- 1430 of the subarray specified in the data lifetime.

1431 2.7.5. copy clause

1432 The **copy** clause may appear on structured **data** and compute constructs and on **declare** direc-

1433 tives.

1434 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,

1435 the **copy** clause behaves as follows:

- 1436 • At entry to the region:
- 1437 – If *var* is present, a *present increment* action with the structured reference counter is
- 1438 performed. If *var* is a pointer reference, an *attach* action is performed.
- 1439 – Otherwise, a *copyin* action with the structured reference counter is performed. If *var* is
- 1440 a pointer reference, an *attach* action is performed.
- 1441 • At exit from the region:
- 1442 – If *var* is not present in the current device memory, a runtime error is issued.
- 1443 – Otherwise, a *present decrement* action with the structured reference counter is per-
- 1444 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
- 1445 and dynamic reference counters are zero, a *copyout* action is performed.

1446 The restrictions regarding subarrays in the **present** clause apply to this clause.

1447 For compatibility with OpenACC 2.0, **present_or_copy** and **pcopy** are alternate names for

1448 **copy**.

1449 2.7.6. copyin clause

1450 The **copyin** clause may appear on structured **data** and compute constructs, on **declare** direc-

1451 tives, and on **enter data** directives.

1452 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,

1453 the **copyin** clause behaves as follows:

- 1454 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
1455 the dynamic reference counter is used.
- 1456 – If *var* is present, a *present increment* action with the appropriate reference counter is
1457 performed. If *var* is a pointer reference, an *attach* action is performed.
- 1458 – Otherwise, a *copyin* action with the appropriate reference counter is performed. If *var*
1459 is a pointer reference, an *attach* action is performed.

- 1460 • At exit from the region:
- 1461 – If *var* is not present in the current device memory, a runtime error is issued.
- 1462 – Otherwise, a *present decrement* action with the structured reference counter is per-
1463 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
1464 and dynamic reference counters are zero, a *delete* action is performed.

1465 If the optional **readonly** modifier appears, then the implementation may assume that the data
1466 referenced by *var-list* is never written to within the applicable region.

1467 The restrictions regarding subarrays in the **present** clause apply to this clause.

1468 For compatibility with OpenACC 2.0, **present_or_copyin** and **pcopyin** are alternate names
1469 for **copyin**.

1470 An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc_copyin**
1471 API routine, as described in Section 3.2.26.

1472 2.7.7. copyout clause

1473 The **copyout** clause may appear on structured **data** and compute constructs, on **declare** di-
1474 rectives, and on **exit data** directives. The clause may optionally have a **zero** modifier if the
1475 **copyout** clause appears on a structured **data** or compute construct.

1476 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1477 the **copyout** clause behaves as follows:

- 1478 • At entry to a region:
- 1479 – If *var* is present, a *present increment* action with the structured reference counter is
1480 performed. If *var* is a pointer reference, an *attach* action is performed.
- 1481 – Otherwise, a *create* action with the structured reference is performed. If *var* is a pointer
1482 reference, an *attach* action is performed. If a **zero** modifier appears, the memory is
1483 zeroed after the *create* action.
- 1484 • At exit from a region, the structured reference counter is used. On an **exit data** directive,
1485 the dynamic reference counter is used.
- 1486 – If *var* is not present in the current device memory, a runtime error is issued.
- 1487 – Otherwise, the reference counter is updated:
- 1488 * On an **exit data** directive with a **finalize** clause, the dynamic reference
1489 counter is set to zero.
- 1490 * Otherwise, a *present decrement* action with the appropriate reference counter is

1491 performed.

1492 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic
1493 reference counters are zero, a *copyout* action is performed.

1494 The restrictions regarding subarrays in the **present** clause apply to this clause.

1495 For compatibility with OpenACC 2.0, **present_or_copyout** and **pcopyout** are alternate
1496 names for **copyout**.

1497 An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is func-
1498 tionally equivalent to a call to the **acc_copyout_finalize** or **acc_copyout** API routine,
1499 respectively, as described in Section 3.2.28.

1500 2.7.8. create clause

1501 The **create** clause may appear on structured **data** and compute constructs, on **declare** direc-
1502 tives, and on **enter data** directives. The clause may optionally have a **zero** modifier.

1503 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1504 the **create** clause behaves as follows:

1505 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
1506 the dynamic reference counter is used.

1507 – If *var* is present, a *present increment* action with the appropriate reference counter is
1508 performed. If *var* is a pointer reference, an *attach* action is performed.

1509 – Otherwise, a *create* action with the appropriate reference counter is performed. If *var*
1510 is a pointer reference, an *attach* action is performed. If a **zero** modifier appears, the
1511 memory is zeroed after the *create* action.

1512 • At exit from the region:

1513 – If *var* is not present in the current device memory, a runtime error is issued.

1514 – Otherwise, a *present decrement* action with the structured reference counter is per-
1515 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
1516 and dynamic reference counters are zero, a *delete* action is performed.

1517 The restrictions regarding subarrays in the **present** clause apply to this clause.

1518 For compatibility with OpenACC 2.0, **present_or_create** and **pcreate** are alternate names
1519 for **create**.

1520 An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc_create**
1521 API routine, as described in Section 3.2.27.

1522 2.7.9. no_create clause

1523 The **no_create** clause may appear on structured **data** and compute constructs.

1524 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1525 the **no_create** clause behaves as follows:

1526 • At entry to the region:

- 1527 – If *var* is present, a *present increment* action with the structured reference counter is
 1528 performed. If *var* is a pointer reference, an *attach* action is performed.
- 1529 – Otherwise, no action is performed, and any device code in this construct will use the
 1530 local memory address for *var*.
- 1531 • At exit from the region:
- 1532 – If *var* is not present in the current device memory, no action is performed.
- 1533 – Otherwise, a *present decrement* action with the structured reference counter is per-
 1534 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
 1535 and dynamic reference counters are zero, a *delete* action is performed.
- 1536 The restrictions regarding subarrays in the **present** clause apply to this clause.

1537 2.7.10. delete clause

- 1538 The **delete** clause may appear on **exit data** directives.
- 1539 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1540 the **delete** clause behaves as follows:
- 1541 • If *var* is not present in the current device memory, a runtime error is issued.
- 1542 • Otherwise, the dynamic reference counter is updated:
- 1543 – On an **exit data** directive with a **finalize** clause, the dynamic reference counter
 1544 is set to zero.
- 1545 – Otherwise, a *present decrement* action with the dynamic reference counter is performed.
- 1546 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic
 1547 reference counters are zero, a *delete* action is performed.
- 1548 An **exit data** directive with a **delete** clause and with or without a **finalize** clause is func-
 1549 tionally equivalent to a call to the **acc_delete_finalize** or **acc_delete** API routine, re-
 1550 spectively, as described in Section 3.2.29.

1551 2.7.11. attach clause

- 1552 The **attach** clause may appear on structured **data** and compute constructs and on **enter data**
 1553 directives. Each *var* argument to an **attach** clause must be a C or C++ pointer or a Fortran variable
 1554 or array with the **pointer** or **allocatable** attribute.
- 1555 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1556 the **attach** clause behaves as follows:
- 1557 • At entry to a region or at an **enter data** directive, an *attach* action is performed.
- 1558 • At exit from the region, a *detach* action is performed.

1559 **2.7.12. detach clause**

1560 The **detach** clause may appear on **exit data** directives. Each *var* argument to a **detach** clause
 1561 must be a C or C++ pointer or a Fortran variable or array with the **pointer** or **allocatable**
 1562 attribute.

1563 For each *var* in *varlist*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1564 the **detach** clause behaves as follows:

- 1565 • If there is a **finalize** clause on the **exit data** directive, an *immediate detach* action is
 1566 performed.
- 1567 • Otherwise, a *detach* action is performed.

1568 **2.8. Host_Data Construct**

1569 **Summary** The **host_data** construct makes the address of data in device memory available on
 1570 the host.

1571 **Syntax** In C and C++, the syntax of the OpenACC **host_data** construct is

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

1572 and in Fortran, the syntax is

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

1573 where *clause* is one of the following:

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

1574 **Description** This construct is used to make the address of data in device memory available in
 1575 host code.

1576 **Restrictions**

- 1577 • A *var* in a **use_device** clause must be the name of a variable or array.
- 1578 • At least one **use_device** clause must appear.
- 1579 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
 1580 value; in C or C++, the condition must evaluate to a scalar integer value.

- 1581 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
1582 **use_device** clauses.

1583 2.8.1. use_device clause

1584 The **use_device** clause tells the compiler to use the current device address of any *var* in *var-list*
1585 in code within the construct. In particular, this may be used to pass the device address of *var* to
1586 optimized procedures written in a lower-level API. When there is no **if_present** clause, and
1587 either there is no **if** clause or the condition in the **if** clause evaluates to nonzero (in C or C++)
1588 or **.true.** (in Fortran), the *var* in *var-list* must be present in the accelerator memory due to data
1589 regions or data lifetimes that contain this construct. For data in shared memory, the device address
1590 is the same as the host address.

1591 2.8.2. if clause

1592 The **if** clause is optional. When an **if** clause appears and the condition evaluates to zero in C
1593 or C++, or **.false.** in Fortran, the compiler will not replace the addresses of any *var* in code
1594 within the construct. When there is no **if** clause, or when an **if** clause appears and the condition
1595 evaluates to nonzero in C or C++, or **.true.** in Fortran, the compiler will replace the addresses as
1596 described in the previous subsection.

1597 2.8.3. if_present clause

1598 When an **if_present** clause appears on the directive, the compiler will only replace the address
1599 of any *var* which appears in *var-list* that is present in the current device memory.

1600 2.9. Loop Construct

1601 **Summary** The OpenACC **loop** construct applies to a loop which must immediately follow this
1602 directive. The **loop** construct can describe what type of parallelism to use to execute the loop and
1603 declare private *vars* and reduction operations.

1604 **Syntax** In C and C++, the syntax of the **loop** construct is

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

1605 In Fortran, the syntax of the **loop** construct is

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

1606 where *clause* is one of the following:

```

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

```

1607 where *gang-arg* is one of:

```

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

```

1608 and *gang-arg-list* may have at most one **num** and one **static** argument,

1609 and where *size-expr* is one of:

```

*
int-expr

```

1610 Some clauses are only valid in the context of a **kernels** construct; see the descriptions below.

1611 An *orphaned loop* construct is a **loop** construct that is not lexically enclosed within a compute
 1612 construct. The parent compute construct of a **loop** construct is the nearest compute construct that
 1613 lexically contains the **loop** construct.

1614 Restrictions

- 1615 • Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **independent**, **auto**, and **tile**
 1616 clauses may follow a **device_type** clause.
- 1617 • The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels
 1618 region.
- 1619 • A loop associated with a **loop** construct that does not have a **seq** clause must be written
 1620 such that the loop iteration count is computable when entering the **loop** construct.
- 1621 • Only one of the **seq**, **independent**, and **auto** clauses may appear.
- 1622 • A **gang**, **worker**, or **vector** clause may not appear if a **seq** clause appears.

1623 2.9.1. collapse clause

1624 The **collapse** clause is used to specify how many tightly nested loops are associated with the
 1625 **loop** construct. The argument to the **collapse** clause must be a constant positive integer expres-

1626 sion. If no **collapse** clause appears, only the immediately following loop is associated with the
1627 **loop** construct.

1628 If more than one loop is associated with the **loop** construct, the iterations of all the associated loops
1629 are all scheduled according to the rest of the clauses. The trip count for all loops associated with the
1630 **collapse** clause must be computable and invariant in all the loops.

1631 It is implementation-defined whether a **gang**, **worker** or **vector** clause on the construct is ap-
1632 plied to each loop, or to the linearized iteration space.

1633 2.9.2. gang clause

1634 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
1635 the **gang** clause specifies that the iterations of the associated loop or loops are to be executed in
1636 parallel by distributing the iterations among the gangs created by the **parallel** construct. A
1637 **loop** construct with the **gang** clause transitions a compute region from gang-redundant mode to
1638 gang-partitioned mode. The number of gangs is controlled by the **parallel** construct; only the
1639 **static** argument is allowed. The loop iterations must be data independent, except for *vars* which
1640 appear in a **reduction** clause or which are modified in an atomic region. The region of a loop
1641 with the **gang** clause may not contain another loop with the **gang** clause unless within a nested
1642 compute region.

1643 When the parent compute construct is a **kernels** construct, the **gang** clause specifies that the
1644 iterations of the associated loop or loops are to be executed in parallel across the gangs. An argument
1645 with no keyword or with the **num** keyword is allowed only when the **num_gangs** does not appear
1646 on the **kernels** construct. If an argument with no keyword or an argument after the **num** keyword
1647 appears, it specifies how many gangs to use to execute the iterations of this loop. The region of a
1648 loop with the **gang** clause may not contain another loop with a **gang** clause unless within a nested
1649 compute region.

1650 The scheduling of loop iterations to gangs is not specified unless the **static** modifier appears as
1651 an argument. If the **static** modifier appears with an integer expression, that expression is used
1652 as a *chunk* size. If the static modifier appears with an asterisk, the implementation will select a
1653 *chunk* size. The iterations are divided into chunks of the selected *chunk* size, and the chunks are
1654 assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops
1655 in the same parallel region with the same number of iterations, and with **static** clauses with the
1656 same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the
1657 same kernels region with the same number of iterations, the same number of gangs to use, and with
1658 **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

1659 2.9.3. worker clause

1660 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
1661 the **worker** clause specifies that the iterations of the associated loop or loops are to be executed
1662 in parallel by distributing the iterations among the multiple workers within a single gang. A **loop**
1663 construct with a **worker** clause causes a gang to transition from worker-single mode to worker-
1664 partitioned mode. In contrast to the **gang** clause, the **worker** clause first activates additional
1665 worker-level parallelism and then distributes the loop iterations across those workers. No argu-
1666 ment is allowed. The loop iterations must be data independent, except for *vars* which appear in

1667 a **reduction** clause or which are modified in an atomic region. The region of a loop with the
1668 **worker** clause may not contain a loop with the **gang** or **worker** clause unless within a nested
1669 compute region.

1670 When the parent compute construct is a **kernels** construct, the **worker** clause specifies that the
1671 iterations of the associated loop or loops are to be executed in parallel across the workers within
1672 a single gang. An argument is allowed only when the **num_workers** does not appear on the
1673 **kernels** construct. The optional argument specifies how many workers per gang to use to execute
1674 the iterations of this loop. The region of a loop with the **worker** clause may not contain a loop
1675 with a **gang** or **worker** clause unless within a nested compute region.

1676 All workers will complete execution of their assigned iterations before any worker proceeds beyond
1677 the end of the loop.

1678 2.9.4. vector clause

1679 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
1680 the **vector** clause specifies that the iterations of the associated loop or loops are to be executed
1681 in vector or SIMD mode. A **loop** construct with a **vector** clause causes a worker to transition
1682 from vector-single mode to vector-partitioned mode. Similar to the **worker** clause, the **vector**
1683 clause first activates additional vector-level parallelism and then distributes the loop iterations across
1684 those vector lanes. The operations will execute using vectors of the length specified or chosen for
1685 the parallel region. The loop iterations must be data independent, except for *vars* which appear in
1686 a **reduction** clause or which are modified in an atomic region. The region of a loop with the
1687 **vector** clause may not contain a loop with the **gang**, **worker**, or **vector** clause unless within
1688 a nested compute region.

1689 When the parent compute construct is a **kernels** construct, the **vector** clause specifies that the
1690 iterations of the associated loop or loops are to be executed with vector or SIMD processing. An
1691 argument is allowed only when the **vector_length** does not appear on the **kernels** construct.
1692 If an argument appears, the iterations will be processed in vector strips of that length; if no argument
1693 appears, the implementation will choose an appropriate vector length. The region of a loop with the
1694 **vector** clause may not contain a loop with a **gang**, **worker**, or **vector** clause unless within a
1695 nested compute region.

1696 All vector lanes will complete execution of their assigned iterations before any vector lane proceeds
1697 beyond the end of the loop.

1698 2.9.5. seq clause

1699 The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the
1700 accelerator. This clause will override any automatic parallelization or vectorization.

1701 2.9.6. auto clause

1702 The **auto** clause specifies that the implementation must analyze the loop and determine whether the
1703 loop iterations are data-independent. If it determines that the loop iterations are data-independent,
1704 the implementation must treat the **auto** clause as if it is an **independent** clause. If not, or if it

1705 is unable to make a determination, it must treat the **auto** clause as if it is a **seq** clause, and it must
1706 ignore any **gang**, **worker**, or **vector** clauses on the loop construct.

1707 When the parent compute construct is a **kernels** construct, a **loop** construct with no **independent**
1708 or **seq** clause is treated as if it has the **auto** clause.

1709 2.9.7. tile clause

1710 The **tile** clause specifies that the implementation should split each loop in the loop nest into two
1711 loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile**
1712 clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression
1713 or an asterisk. If there are n tile sizes in the list, the **loop** construct must be immediately followed
1714 by n tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop
1715 of the n associated loops, and the last element corresponds to the outermost associated loop. If the
1716 tile size is an asterisk, the implementation will choose an appropriate value. Each loop in the nest
1717 will be split or *strip-mined* into two loops, an outer *tile* loop and an inner *element* loop. The trip
1718 count of the element loop will be limited to the corresponding tile size from the *size-expr-list*. The
1719 *tile* loops will be reordered to be outside all the *element* loops, and the *element* loops will all be
1720 inside the *tile* loops.

1721 If the **vector** clause appears on the **loop** construct, the **vector** clause is applied to the *element*
1722 loops. If the **gang** clause appears on the **loop** construct, the **gang** clause is applied to the *tile*
1723 loops. If the **worker** clause appears on the **loop** construct, the **worker** clause is applied to the
1724 *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

1725 2.9.8. device_type clause

1726 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

1727 2.9.9. independent clause

1728 The **independent** clause tells the implementation that the loop iterations must be data indepen-
1729 dent, except for *vars* which appear in a **reduction** clause or which are modified in an atomic
1730 region. This allows the implementation to generate code to execute the iterations in parallel with no
1731 synchronization.

1732 A **loop** construct with no **auto** or **seq** clause is treated as if it has the **independent** clause
1733 when it is an orphaned **loop** construct or its parent compute construct is a **parallel** construct.

1734 Note

- 1735 • It is likely a programming error to use the **independent** clause on a loop if any iteration
1736 writes to a variable or array element that any other iteration also writes or reads, except for
1737 *vars* which appear in a **reduction** clause or which are modified in an atomic region.
- 1738 • The implementation may be restricted in the levels of parallelism it can apply by the presence
1739 of **loop** constructs with **gang**, **worker**, or **vector** clauses for outer or inner loops.

1740 2.9.10. private clause

1741 The **private** clause on a **loop** construct specifies that a copy of each item in *var-list* will be
1742 created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created
1743 for each thread associated with each vector lane. If the body of the loop is executed in *worker-*
1744 *partitioned vector-single* mode, a copy of the item is created for and shared across the set of threads
1745 associated with all the vector lanes of each worker. Otherwise, a copy of the item is created for and
1746 shared across the set of threads associated with all the vector lanes of all the workers of each gang.

1747 Restrictions

- 1748 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
1749 **private** clauses.

1750 2.9.11. reduction clause

1751 The **reduction** clause specifies a reduction operator and one or more *vars*. For each reduction
1752 *var*, a private copy is created in the same manner as for a **private** clause on the **loop** construct,
1753 and initialized for that operator; see the table in Section 2.5.13 reduction clause. After the loop, the
1754 values for each thread are combined using the specified reduction operator, and the result combined
1755 with the value of the original *var* and stored in the original *var*. If the original *var* is not private,
1756 this update occurs by the end of the compute region, and any access to the original *var* is undefined
1757 within the compute region. Otherwise, the update occurs at the end of the loop. If the reduction
1758 *var* is an array or subarray, the reduction operation is logically equivalent to applying that reduction
1759 operation to each array element of the array or subarray individually. If the reduction *var* is a com-
1760 posite variable, the reduction operation is logically equivalent to applying that reduction operation
1761 to each member of the composite variable individually.

1762 If a variable is involved in a reduction that spans multiple nested loops where two or more of those
1763 loops have associated **loop** directives, a **reduction** clause containing that variable must appear
1764 on each of those **loop** directives.

1765 Restrictions

- 1766 • A *var* in a **reduction** clause must be a scalar variable name, a composite variable name,
1767 an array name, an array element, or a subarray (refer to Section 2.7.1).
- 1768 • Reduction clauses on nested constructs for the same reduction *var* must have the same reduc-
1769 tion operator.
- 1770 • Every *var* in a **reduction** clause appearing on an orphaned **loop** construct must be private.
- 1771 • The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.13
1772 reduction clause also apply to a **reduction** clause on a loop construct.
- 1773 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
1774 **reduction** clauses.

1775

1776 **Examples**

- 1777 • **x** is not private at the **loop** directive below, so its reduction normally updates **x** at the end
 1778 of the parallel region, where gangs synchronize. When possible, the implementation might
 1779 choose to partially update **x** at the loop exit instead, or fully if **num_gangs(1)** were added
 1780 to the **parallel** directive. However, portable applications cannot rely on such early up-
 1781 dates, so accesses to **x** are undefined within the parallel region outside the loop.

```

1782     int x = 0;
1783     #pragma acc parallel copy(x)
1784     {
1785         // gang-shared x undefined
1786         #pragma acc loop gang worker vector reduction(+:x)
1787         for (int i = 0; i < I; ++i)
1788             x += 1; // vector-private x modified
1789         // gang-shared x undefined
1790     } // gang-shared x updated for gang/worker/vector reduction
1791     // x = I
  
```

- 1792 • **x** is private at each of the innermost two **loop** directives below, so each of their reductions
 1793 updates **x** at the loop's exit. However, **x** is not private at the outer **loop** directive, so its
 1794 reduction updates **x** by the end of the parallel region instead.

```

1795     int x = 0;
1796     #pragma acc parallel copy(x)
1797     {
1798         // gang-shared x undefined
1799         #pragma acc loop gang reduction(+:x)
1800         for (int i = 0; i < I; ++i) {
1801             #pragma acc loop worker reduction(+:x)
1802             for (int j = 0; j < J; ++j) {
1803                 #pragma acc loop vector reduction(+:x)
1804                 for (int k = 0; k < K; ++k) {
1805                     x += 1; // vector-private x modified
1806                 } // worker-private x updated for vector reduction
1807             } // gang-private x updated for worker reduction
1808         }
1809         // gang-shared x undefined
1810     } // gang-shared x updated for gang reduction
1811     // x = I * J * K
  
```

- 1812 • At each **loop** directive below, **x** is private due to its implicit **firstprivate** attribute on
 1813 the **parallel** directive, but **y** is not private due to its **copy** clause on the **parallel**
 1814 directive. Thus, each reduction updates **x** at the loop exit, but each reduction updates **y** by
 1815 the end of the parallel region instead.

```

1816     int x = 0, y = 0;
1817     #pragma acc parallel copy(y) // firstprivate(x) implied
1818     {
  
```

```

1819 // gang-private x = 0; gang-shared y undefined
1820 #pragma acc loop seq reduction(+:x,y)
1821 for (int i = 0; i < I; ++i) {
1822     x += 1; y += 2; // loop-private x and y modified
1823 } // gang-private x updated for seq reduction (trivial reduction)
1824 // gang-private x = I; gang-shared y undefined
1825 #pragma acc loop worker reduction(+:x,y)
1826 for (int i = 0; i < I; ++i) {
1827     x += 1; y += 2; // worker-private x and y modified
1828 } // gang-private x updated for worker reduction
1829 // gang-private x = 2 * I; gang-shared y undefined
1830 #pragma acc loop vector reduction(+:x,y)
1831 for (int i = 0; i < I; ++i) {
1832     x += 1; y += 2; // vector-private x and y modified
1833 } // gang-private x updated for vector reduction
1834 // gang-private x = 3 * I; gang-shared y undefined
1835 } // gang-shared y updated for gang/seq/worker/vector reductions
1836 // x = 0; y = 3 * I * 2

```

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

```

1841 int x = 0;
1842 #pragma acc parallel loop worker reduction(+:x)
1843 for (int i = 0; i < I; ++i) {
1844     x += 1; // worker-private x modified
1845 } // gang-shared x updated for gang/worker reduction
1846 // x = I
1847
1848 int x = 0;
1849 #pragma acc parallel copy(x)
1850 {
1851     // gang-shared x undefined
1852     #pragma acc loop worker reduction(+:x)
1853     for (int i = 0; i < I; ++i) {
1854         x += 1; // worker-private x modified
1855     }
1856     // gang-shared x undefined
1857 } // gang-shared x updated for gang/worker reduction
1858 // x = I

```

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

```

1864     int x = 0;
1865     const int *arr = /*array of I values*/;
1866     #pragma acc parallel copy(x)
1867     {
1868         // gang-shared x undefined
1869         #pragma acc loop auto gang reduction(max:x)
1870         for (int i = 0; i < I; ++i) {
1871             // complex loop body
1872             x = x < arr[i] ? arr[i] : x; // gang or loop-private x modified
1873         }
1874         // gang-shared x undefined
1875     } // gang-shared x updated for gang or gang/seq reduction
1876     // x = arr maximum

```

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

```

1882     int x = 0;
1883     const int *arr = /*array of I values*/;
1884     #pragma acc parallel copy(x)
1885     {
1886         // gang-shared x undefined
1887         #pragma acc loop auto gang reduction(+:x)
1888         for (int i = 0; i < I; ++i) {
1889             // complex loop body
1890             x += arr[i]; // gang or loop-private x modified
1891         }
1892         // gang-shared x undefined
1893     } // gang-shared x updated for gang or gang/seq reduction
1894     // x = arr sum possibly times number of gangs

```

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

```

1900     int x = 0, y = 0;
1901     #pragma acc parallel copy(x) reduction(+:x,y)
1902     {
1903         int z = 0;
1904         #pragma acc loop gang vector reduction(+:x,z)
1905         for (int i = 0; i < I; ++i) {
1906             x += 1; z += 2; // vector-private x and z modified
1907         } // gang-private x and z updated for vector reduction (trivial 1-gang reduction)
1908         y += z; // gang-private y modified

```

```

1909     } // gang-shared x and y updated for gang reduction
1910     // x = I; y = I * 2

```

```

1911 ▲
1912

```

1913 2.10. Cache Directive

1914 **Summary** The **cache** directive may appear at the top of (inside of) a loop. It specifies array
 1915 elements or subarrays that should be fetched into the highest level of the cache for the body of the
 1916 loop.

1917 **Syntax** In C and C++, the syntax of the **cache** directive is

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

1918 In Fortran, the syntax of the **cache** directive is

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

1919 A *var* in a **cache** directive must be a single array element or a simple subarray. In C and C++,
 1920 a simple subarray is an array name followed by an extended array range specification in brackets,
 1921 with start and length, such as

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

1922 where the lower bound is a constant, loop invariant, or the **for** loop index variable plus or minus a
 1923 constant or loop invariant, and the length is a constant.

1924 In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifi-
 1925 cations in parentheses, with lower and upper bound subscripts, such as

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

1926 The lower bounds must be constant, loop invariant, or the **do** loop index variable plus or minus
 1927 a constant or loop invariant; moreover the difference between the corresponding upper and lower
 1928 bounds must be a constant.

1929 If the optional **readonly** modifier appears, then the implementation may assume that the data
 1930 referenced by any *var* in that directive is never written to within the applicable region.

1931 Restrictions

- 1932 • If an array element or subarray is listed in a **cache** directive, all references to that array
 1933 during execution of that loop iteration must not refer to elements of the array outside the
 1934 index range specified in the **cache** directive.
- 1935 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
 1936 **cache** directives.

1937 2.11. Combined Constructs

1938 **Summary** The combined OpenACC **parallel loop**, **kernels loop**, and **serial loop**
 1939 constructs are shortcuts for specifying a **loop** construct nested immediately inside a **parallel**,
 1940 **kernels**, or **serial** construct. The meaning is identical to explicitly specifying a **parallel**,
 1941 **kernels**, or **serial** construct containing a **loop** construct. Any clause that is allowed on a
 1942 **parallel** or **loop** construct is allowed on the **parallel loop** construct; any clause allowed
 1943 on a **kernels** or **loop** construct is allowed on a **kernels loop** construct; and any clause
 1944 allowed on a **serial** or **loop** construct is allowed on a **serial loop** construct.

1945 **Syntax** In C and C++, the syntax of the **parallel loop** construct is

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

1946 In Fortran, the syntax of the **parallel loop** construct is

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

1947 The associated structured block is the loop which must immediately follow the directive. Any of
 1948 the **parallel** or **loop** clauses valid in a parallel region may appear.

1949 In C and C++, the syntax of the **kernels loop** construct is

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

1950 In Fortran, the syntax of the **kernels loop** construct is

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

1951 The associated structured block is the loop which must immediately follow the directive. Any of
 1952 the **kernels** or **loop** clauses valid in a kernels region may appear.

1953 In C and C++, the syntax of the **serial loop** construct is

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

1954 In Fortran, the syntax of the **serial loop** construct is

```

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

```

1955 The associated structured block is the loop which must immediately follow the directive. Any of
 1956 the **serial** or **loop** clauses valid in a serial region may appear.

1957 A **private** or **reduction** clause on a combined construct is treated as if it appeared on the
 1958 **loop** construct. In addition, a **reduction** clause on a combined construct implies a **copy** data
 1959 clause for each reduction variable, unless a data clause for that variable appears on the combined
 1960 construct.

1961 Restrictions

- 1962 • The restrictions for the **parallel**, **kernels**, **serial**, and **loop** constructs apply.

1963 2.12. Atomic Construct

1964 **Summary** An **atomic** construct ensures that a specific storage location is accessed and/or up-
 1965 dated atomically, preventing simultaneous reading and writing by gangs, workers, and vector threads
 1966 that could result in indeterminate values.

1967 **Syntax** In C and C++, the syntax of the **atomic** constructs is:

```

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

```

1968 OR:

```

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

```

1969 Where *atomic-clause* is one of **read**, **write**, **update**, or **capture**. The *expression-stmt* is an
 1970 expression statement with one of the following forms:

1971 If the *atomic-clause* is **read**:

```

v = x;

```

1972 If the *atomic-clause* is **write**:

```

x = expr;

```

1973 If the *atomic-clause* is **update** or no clause appears:

```

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

```

1974 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;

```

1975 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;}

```

1976 In the preceding expressions:

- 1977 • **x** and **v** (as applicable) are both l-value expressions with scalar type.
- 1978 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
1979 the same storage location.
- 1980 • Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- 1981 • Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.
- 1982 • *expr* is an expression with scalar type.
- 1983 • *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.
- 1984 • *binop*, *binop=*, **++**, and **--** are not overloaded operators.

- 1985 • The expression $\mathbf{x} \text{ binop } \text{expr}$ must be mathematically equivalent to $\mathbf{x} \text{ binop } (\text{expr})$. This
- 1986 requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by
- 1987 using parentheses around *expr* or subexpressions of *expr*.
- 1988 • The expression $\text{expr binop } \mathbf{x}$ must be mathematically equivalent to $(\text{expr}) \text{ binop } \mathbf{x}$. This
- 1989 requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*,
- 1990 or by using parentheses around *expr* or subexpressions of *expr*.
- 1991 • For forms that allow multiple occurrences of \mathbf{x} , the number of times that \mathbf{x} is evaluated is
- 1992 unspecified.

1993 In Fortran the syntax of the **atomic** constructs is:

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

1994 OR

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

1995 OR

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

1996 OR

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

1997 OR

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

1998 OR

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

1999 where *write-statement* has the following form (if *atomic-clause* is **write** or **capture**):

$$\mathbf{x} = \mathbf{expr}$$

2000 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

$$\mathbf{v} = \mathbf{x}$$

2001 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**,
2002 or no clause appears):

$$\begin{aligned} \mathbf{x} &= \mathbf{x} \operatorname{operator} \mathbf{expr} \\ \mathbf{x} &= \mathbf{expr} \operatorname{operator} \mathbf{x} \\ \mathbf{x} &= \operatorname{intrinsic_procedure_name}(\mathbf{x}, \mathbf{expr_list}) \\ \mathbf{x} &= \operatorname{intrinsic_procedure_name}(\mathbf{expr_list}, \mathbf{x}) \end{aligned}$$

2003 In the preceding statements:

- 2004 • \mathbf{x} and \mathbf{v} (as applicable) are both scalar variables of intrinsic type.
- 2005 • \mathbf{x} must not be an allocatable variable.
- 2006 • During the execution of an atomic region, multiple syntactic occurrences of \mathbf{x} must designate
2007 the same storage location.
- 2008 • None of \mathbf{v} , \mathbf{expr} , and $\mathbf{expr_list}$ (as applicable) may access the same storage location as \mathbf{x} .
- 2009 • None of \mathbf{x} , \mathbf{expr} , and $\mathbf{expr_list}$ (as applicable) may access the same storage location as \mathbf{v} .
- 2010 • \mathbf{expr} is a scalar expression.
- 2011 • $\mathbf{expr_list}$ is a comma-separated, non-empty list of scalar expressions. If *intrinsic_procedure_name*
2012 refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in $\mathbf{expr_list}$.
- 2013 • *intrinsic_procedure_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**,
2014 *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**
- 2015 • The expression $\mathbf{x} \operatorname{operator} \mathbf{expr}$ must be mathematically equivalent to $\mathbf{x} \operatorname{operator} (\mathbf{expr})$.
2016 This requirement is satisfied if the operators in \mathbf{expr} have precedence greater than *operator*,
2017 or by using parentheses around \mathbf{expr} or subexpressions of \mathbf{expr} .
- 2018 • The expression $\mathbf{expr} \operatorname{operator} \mathbf{x}$ must be mathematically equivalent to $(\mathbf{expr}) \operatorname{operator} \mathbf{x}$.
2019 This requirement is satisfied if the operators in \mathbf{expr} have precedence equal to or greater than
2020 *operator*, or by using parentheses around \mathbf{expr} or subexpressions of \mathbf{expr} .
- 2021 • *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other program
2022 entities.
- 2023 • *operator* must refer to the intrinsic operator and not to a user-defined operator. All assign-
2024 ments must be intrinsic assignments.

- For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is unspecified.

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

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

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

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

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

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

Restrictions

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

2.13. Declare Directive

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

2064 **Syntax** In C and C++, the syntax of the **declare** directive is:

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

2065 In Fortran the syntax of the **declare** directive is:

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

2066 where *clause* is one of the following:

```
copy( var-list )
copyin( [readonly:]var-list )
copyout( var-list )
create( var-list )
present( var-list )
deviceptr( var-list )
device_resident( var-list )
link( var-list )
```

2067 The associated region is the implicit region associated with the function, subroutine, or program in
 2068 which the directive appears. If the directive appears in the declaration section of a Fortran *module*
 2069 subprogram or in a C or C++ global scope, the associated region is the implicit region for the whole
 2070 program. The **copy**, **copyin**, **copyout**, **present**, and **deviceptr** data clauses are described
 2071 in Section 2.7 Data Clauses.

2072 Restrictions

- 2073 • A **declare** directive must appear in the same scope as any *var* in any of the data clauses on
 2074 the directive.
- 2075 • At least one clause must appear on a **declare** directive.
- 2076 • A *var* in a **declare** declare must be a variable or array name, or a Fortran *common block*
 2077 name between slashes.
- 2078 • A *var* may appear at most once in all the clauses of **declare** directives for a function,
 2079 subroutine, program, or module.
- 2080 • In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.
- 2081 • In Fortran, pointer arrays may appear, but pointer association is not preserved in device mem-
 2082 ory.
- 2083 • In a Fortran *module* declaration section, only **create**, **copyin**, **device_resident**, and
 2084 **link** clauses are allowed.
- 2085 • In C or C++ global scope, only **create**, **copyin**, **deviceptr**, **device_resident** and
 2086 **link** clauses are allowed.
- 2087 • C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**, **device_resident**
 2088 and **link** clauses on a **declare** directive.

- 2089 • In C and C++, only global and *extern* variables may appear in a **link** clause. In Fortran,
2090 only *module* variables and *common* block names (enclosed in slashes) may appear in a **link**
2091 clause.
- 2092 • In C or C++, a **longjmp** call in the region must return to a **setjmp** call within the region.
- 2093 • In C++, an exception thrown in the region must be handled within the region.
- 2094 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional dummy ar-
2095 guments in data clauses, including **device_resident** clauses.

2096 2.13.1. device_resident clause

2097 **Summary** The **device_resident** clause specifies that the memory for the named variables
2098 should be allocated in the current device memory and not in local memory. The host may not be
2099 able to access variables in a **device_resident** clause. The accelerator data lifetime of global
2100 variables or common blocks that appear in a **device_resident** clause is the entire execution of
2101 the program.

2102 In Fortran, if the variable has the Fortran *allocatable* attribute, the memory for the variable will
2103 be allocated in and deallocated from the current device memory when the host thread executes
2104 an **allocate** or **deallocate** statement for that variable, if the current device is a non-shared
2105 memory device. If the variable has the Fortran *pointer* attribute, it may be allocated or deallocated
2106 by the host in the current device memory, or may appear on the left hand side of a pointer assignment
2107 statement, if the right hand side variable itself appears in a **device_resident** clause.

2108 In Fortran, the argument to a **device_resident** clause may be a *common block* name enclosed
2109 in slashes; in this case, all declarations of the common block must have a matching **device_resident**
2110 clause. In this case, the *common block* will be statically allocated in device memory, and not
2111 in local memory. The *common block* will be available to accelerator routines; see Section 2.15
2112 Procedure Calls in Compute Regions.

2113 In a Fortran *module* declaration section, a *var* in a **device_resident** clause will be available to
2114 accelerator subprograms.

2115 In C or C++ global scope, a *var* in a **device_resident** clause will be available to accelerator
2116 routines. A C or C++ *extern* variable may appear in a **device_resident** clause only if the
2117 actual declaration and all *extern* declarations are also followed by **device_resident** clauses.

2118 2.13.2. create clause

2119 For data in shared memory, no action is taken.

2120 For data not in shared memory, the **create** clause on a **declare** directive behaves as follows,
2121 for each *var* in *var-list*:

- 2122 • At entry to an implicit data region where the **declare** directive appears:
 - 2123 – If *var* is present, a *present increment* action with the structured reference counter is
2124 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 2125 – Otherwise, a *create* action with the structured reference counter is performed. If *var* is
2126 a pointer reference, an *attach* action is performed.

- 2127 • At exit from an implicit data region where the **declare** directive appears:
- 2128 – If *var* is not present in the current device memory, a runtime error is issued.
- 2129 – Otherwise, a *present decrement* action with the structured reference counter is per-
- 2130 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
- 2131 and dynamic reference counters are zero, a *delete* action is performed.

2132 If the **declare** directive appears in a global context, then the data in *var-list* is statically allocated

2133 in device memory and the structured reference counter is set to one.

2134 In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then:

- 2135 • An **allocate** statement for *var* will allocate memory in both local memory as well as in the
- 2136 current device memory, for a non-shared memory device, and the dynamic reference counter
- 2137 will be set to one.
- 2138 • A **deallocate** statement for *var* will deallocate memory from both local memory as well
- 2139 as the current device memory, for a non-shared memory device, and the dynamic reference
- 2140 counter will be set to zero. If the structured reference counter is not zero, a runtime error is
- 2141 issued.

2142 In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the

2143 left hand side of a pointer assignment statement, if the right hand side variable itself appears in a

2144 **create** clause.

2145 2.13.3. link clause

2146 The **link** clause is used for large global host static data that is referenced within an accelerator

2147 routine and that should have a dynamic data lifetime on the device. The **link** clause specifies that

2148 only a global link for the named variables should be statically created in accelerator memory. The

2149 host data structure remains statically allocated and globally available. The device data memory will

2150 be allocated only when the global variable appears on a data clause for a **data** construct, compute

2151 construct, or **enter data** directive. The arguments to the **link** clause must be global data. In C

2152 or C++, the **link** clause must appear at global scope, or the arguments must be *extern* variables.

2153 In Fortran, the **link** clause must appear in a *module* declaration section, or the arguments must be

2154 *common block* names enclosed in slashes. A *common block* that is listed in a **link** clause must be

2155 declared with the same size in all program units where it appears. A **declare link** clause must be

2156 visible everywhere the global variables or common block variables are explicitly or implicitly used

2157 in a data clause, compute construct, or accelerator routine. The global variable or *common block*

2158 variables may be used in accelerator routines. The accelerator data lifetime of variables or common

2159 blocks that appear in a **link** clause is the data region that allocates the variable or common block

2160 with a data clause, or from the execution of the **enter data** directive that allocates the data until

2161 an **exit data** directive deallocates it or until the end of the program.

2162 2.14. Executable Directives

2163 2.14.1. Init Directive

2164 **Summary** The **init** directive tells the runtime to initialize the runtime for that device type.
 2165 This can be used to isolate any initialization cost from the computational cost, when collecting
 2166 performance statistics. If no device type appears all devices will be initialized. An **init** directive
 2167 may be used in place of a call to the **acc_init** runtime API routine, as described in Section 3.2.7.

2168 **Syntax** In C and C++, the syntax of the **init** directive is:

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

2169 In Fortran the syntax of the **init** directive is:

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

2170 where *clause* is one of the following:

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

2171 device_type clause

2172 The **device_type** clause specifies the type of device that is to be initialized in the runtime. If the
 2173 **device_type** clause appears, then the *acc-current-device-type-var* for the current thread is set to
 2174 the argument value. If no **device_num** clause appears then all devices of this type are initialized.

2175 device_num clause

2176 The **device_num** clause specifies the device id to be initialized. If the **device_num** clause
 2177 appears, then the *acc-current-device-num-var* for the current thread is set to the argument value. If
 2178 no **device_type** clause appears, then the specified device id will be initialized for all available
 2179 device types.

2180 if clause

2181 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 2182 perform the initialization unconditionally. When an **if** clause appears, the implementation will gen-
 2183 erate code to conditionally perform the initialization only when the *condition* evaluates to nonzero
 2184 in C or C++, or **.true.** in Fortran.

2185 **Restrictions**

- 2186 • This directive may not be called within a compute region.
- 2187 • If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.
- 2188
- 2189 • If the directive is called more than once without an intervening **acc_shutdown** call or
- 2190 **shutdown** directive, with a different value for the device type argument, the behavior is
- 2191 implementation-defined.
- 2192 • If some accelerator regions are compiled to only use one device type, using this directive with
- 2193 a different device type may produce undefined behavior.

2194 **2.14.2. Shutdown Directive**

2195 **Summary** The **shutdown** directive tells the runtime to shut down the connection to the given
 2196 accelerator, and free any runtime resources. A **shutdown** directive may be used in place of a call
 2197 to the **acc_shutdown** runtime API routine, as described in Section 3.2.8.

2198 **Syntax** In C and C++, the syntax of the **shutdown** directive is:

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

2199 In Fortran the syntax of the **shutdown** directive is:

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

2200 where *clause* is one of the following:

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

2201 **device_type clause**

2202 The **device_type** clause specifies the type of device that is to be disconnected from the runtime.
 2203 If no **device_num** clause appears then all devices of this type are disconnected.

2204 **device_num clause**

2205 The **device_num** clause specifies the device id to be disconnected.
 2206 If no clauses appear then all available devices will be disconnected.

2207 **if clause**

2208 The **if** clause is optional; when there is no **if** clause, the implementation will generate code
 2209 to perform the shutdown unconditionally. When an **if** clause appears, the implementation will
 2210 generate code to conditionally perform the shutdown only when the *condition* evaluates to nonzero
 2211 in C or C++, or **.true.** in Fortran.

2212 **Restrictions**

- 2213 • This directive may not be used during the execution of a compute region.

2214 **2.14.3. Set Directive**

2215 **Summary** The **set** directive provides a means to modify internal control variables using direc-
 2216 tives. Each form of the **set** directive is functionally equivalent to a matching runtime API routine.

2217 **Syntax** In C and C++, the syntax of the **set** directive is:

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

2218 In Fortran the syntax of the **set** directive is:

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

2219 where *clause* is one of the following

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

2220 **default_async clause**

2221 The **default_async** clause specifies the asynchronous queue that should be used if no queue ap-
 2222 pears and changes the value of *acc-default-async-var* for the current thread to the argument value.
 2223 If the value is **acc_async_default**, the value of *acc-default-async-var* will revert to the ini-
 2224 tial value, which is implementation-defined. A **set default_async** directive is functionally
 2225 equivalent to a call to the **acc_set_default_async** runtime API routine, as described in Sec-
 2226 tion 3.2.22.

2227 **device.num clause**

2228 The **device_num** clause specifies the device number to set as the default device for accelerator
 2229 regions and changes the value of *acc-current-device-num-var* for the current thread to the argument

2230 value. If the value of **device_num** argument is negative, the runtime will revert to the default be-
 2231 havior, which is implementation-defined. A **set device_num** directive is functionally equivalent
 2232 to the **acc_set_device_num** runtime API routine, as described in Section 3.2.4.

2233 **device_type** clause

2234 The **device_type** clause specifies the device type to set as the default device type for accelerator
 2235 regions and sets the value of *acc-current-device-type-var* for the current thread to the argument
 2236 value. If the value of the **device_type** argument is zero or the clause does not appear, the
 2237 selected device number will be used for all attached accelerator types. A **set device_type**
 2238 directive is functionally equivalent to a call to the **acc_set_device_type** runtime API routine,
 2239 as described in Section 3.2.2.

2240 **if** clause

2241 The **if** clause is optional; when there is no **if** clause, the implementation will generate code
 2242 to perform the set operation unconditionally. When an **if** clause appears, the implementation
 2243 will generate code to conditionally perform the set operation only when the *condition* evaluates to
 2244 nonzero in C or C++, or **.true.** in Fortran.

2245 **Restrictions**

- 2246 • This directive may not be used within a compute region.
- 2247 • Passing **default_async** the value of **acc_async_noval** has no effect.
- 2248 • Passing **default_async** the value of **acc_async_sync** will cause all asynchronous
 2249 directives in the default asynchronous queue to become synchronous.
- 2250 • Passing **default_async** the value of **acc_async_default** will restore the default
 2251 asynchronous queue to the initial value, which is implementation-defined.
- 2252 • If the value of **device_num** is larger than the maximum supported value for the given type,
 2253 the behavior is implementation-defined.
- 2254 • At least one **default_async**, **device_num**, or **device_type** clause must appear.
- 2255 • Two instances of the same clause may not appear on the same directive.

2256 **2.14.4. Update Directive**

2257 **Summary** The **update** directive is used during the lifetime of accelerator data to update *vars*
 2258 in local memory with values from the corresponding data in device memory, or to update *vars* in
 2259 device memory with values from the corresponding data in local memory.

2260 **Syntax** In C and C++, the syntax of the **update** directive is:

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

2261 In Fortran the syntax of the **update** data directive is:

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

2262 where *clause* is one of the following:

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

2263 Multiple subarrays of the same array may appear in a *var-list* of the same or different clauses on
 2264 the same directive. The effect of an **update** clause is to copy data from device memory to local
 2265 memory for **update self**, and from local memory to device memory for **update device**. The
 2266 updates are done in the order in which they appear on the directive.

2267 Restrictions

- 2268 • At least one **self**, **host**, or **device** clause must appear on an **update** directive.

2269 self clause

2270 The **self** clause specifies that the *vars* in *var-list* are to be copied from the current device memory
 2271 to local memory for data not in shared memory. For data in shared memory, no action is taken. An
 2272 **update** directive with the **self** clause is equivalent to a call to the **acc_update_self** routine,
 2273 described in Section 3.2.31.

2274 host clause

2275 The **host** clause is a synonym for the **self** clause.

2276 device clause

2277 The **device** clause specifies that the *vars* in *var-list* are to be copied from local memory to the cur-
 2278 rent device memory, for data not in shared memory. For data in shared memory, no action is taken.
 2279 An **update** directive with the **device** clause is equivalent to a call to the **acc_update_device**
 2280 routine, described in Section 3.2.30.

2281 **if clause**

2282 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
2283 perform the updates unconditionally. When an **if** clause appears, the implementation will generate
2284 code to conditionally perform the updates only when the *condition* evaluates to nonzero in C or
2285 C++, or **.true.** in Fortran.

2286 **async clause**

2287 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2288 **wait clause**

2289 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2290 **if_present clause**

2291 When an **if_present** clause appears on the directive, no action is taken for a *var* which appears
2292 in *var-list* that is not present in the current device memory. When no **if_present** clause ap-
2293 pears, all *vars* in a **device** or **self** clause must be present in the current device memory, and an
2294 implementation may halt the program with an error message if some data is not present.

2295 **Restrictions**

- 2296 • The **update** directive is executable. It must not appear in place of the statement following
2297 an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical
2298 *if* in Fortran.
- 2299 • If no **if_present** clause appears on the directive, each *var* in *var-list* must be present in
2300 the current device memory.
- 2301 • Only the **async** and **wait** clauses may follow a **device_type** clause.
- 2302 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
2303 value; in C or C++, the condition must evaluate to a scalar integer value.
- 2304 • Noncontiguous subarrays may appear. It is implementation-specific whether noncontiguous
2305 regions are updated by using one transfer for each contiguous subregion, or whether the non-
2306 contiguous data is packed, transferred once, and unpacked, or whether one or more larger
2307 subarrays (no larger than the smallest contiguous region that contains the specified subarray)
2308 are updated.
- 2309 • In C and C++, a member of a struct or class may appear, including a subarray of a member.
2310 Members of a subarray of struct or class type may not appear.
- 2311 • In C and C++, if a subarray notation is used for a struct member, subarray notation may not
2312 be used for any parent of that struct member.
- 2313 • In Fortran, members of variables of derived type may appear, including a subarray of a mem-
2314 ber. Members of subarrays of derived type may not appear.

- 2315 • In Fortran, if array or subarray notation is used for a derived type member, array or subarray
2316 notation may not be used for a parent of that derived type member.
- 2317 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
2318 **self**, **host**, and **device** clauses.

2319 **2.14.5. Wait Directive**

2320 See Section 2.16 Asynchronous Behavior for more information.

2321 **2.14.6. Enter Data Directive**

2322 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2323 **2.14.7. Exit Data Directive**

2324 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2325 **2.15. Procedure Calls in Compute Regions**

2326 This section describes how routines are compiled for an accelerator and how procedure calls are
2327 compiled in compute regions. See Section 2.17 Fortran Optional Arguments for discussion of For-
2328 tran optional arguments in procedure calls inside compute regions.

2329 **2.15.1. Routine Directive**

2330 **Summary** The **routine** directive is used to tell the compiler to compile a given procedure or
2331 a C++ *lambda* for an accelerator as well as for the host. In a file or routine with a procedure call,
2332 the **routine** directive tells the implementation the attributes of the procedure when called on the
2333 accelerator.

2334 **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
```

2335 In C and C++, the **routine** directive without a name may appear immediately before a function
2336 definition, a C++ *lambda*, or just before a function prototype and applies to that immediately fol-
2337 lowing function or prototype. The **routine** directive with a name may appear anywhere that a
2338 function prototype is allowed and applies to the function or the C++ *lambda* in that scope with that
2339 name, but must appear before any definition or use of that function.

2340 In Fortran the syntax of the **routine** directive is:

```

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

```

2341 In Fortran, the **routine** directive without a name may appear within the specification part of a
 2342 subroutine or function definition, or within an interface body for a subroutine or function in an
 2343 interface block, and applies to the containing subroutine or function. The **routine** directive with
 2344 a name may appear in the specification part of a subroutine, function or module, and applies to the
 2345 named subroutine or function.

2346 A C or C++ function or Fortran subprogram compiled with the **routine** directive for an accelera-
 2347 tor is called an *accelerator routine*.

2348 If an *accelerator routine* is a C++ *lambda*, the associated function will be compiled for both the
 2349 accelerator and the host.

2350 If a *lambda* is called in a compute region and it is not an *accelerator routine*, then the *lambda* is
 2351 treated as if its name appears in the name list of a **routine** directive with **seq** clause. If *lambda*
 2352 is defined in an *accelerator routine* that has a **nohost** clause then the *lambda* is treated as if its
 2353 name appears in the name list of a **routine** directive with a **nohost** clause.

2354 The *clause* is one of the following:

```

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

```

2355 A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

2356 **gang clause**

2357 The **gang** clause specifies that the procedure contains, may contain, or may call another procedure
 2358 that contains a loop with a **gang** clause. A call to this procedure must appear in code that is
 2359 executed in *gang-redundant* mode, and all gangs must execute the call. For instance, a procedure
 2360 with a **routine gang** directive may not be called from within a loop that has a **gang** clause.
 2361 Only one of the **gang**, **worker**, **vector** and **seq** clauses may appear for each device type.

2362 **worker clause**

2363 The **worker** clause specifies that the procedure contains, may contain, or may call another pro-
 2364 cedure that contains a loop with a **worker** clause, but does not contain nor does it call another
 2365 procedure that contains a loop with the **gang** clause. A loop in this procedure with an **auto** clause
 2366 may be selected by the compiler to execute in **worker** or **vector** mode. A call to this procedure
 2367 must appear in code that is executed in *worker-single* mode, though it may be in *gang-redundant*

2368 or *gang-partitioned* mode. For instance, a procedure with a **routine worker** directive may be
2369 called from within a loop that has the **gang** clause, but not from within a loop that has the **worker**
2370 clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may appear for each device
2371 type.

2372 **vector clause**

2373 The **vector** clause specifies that the procedure contains, may contain, or may call another pro-
2374 cedure that contains a loop with the **vector** clause, but does not contain nor does it call another
2375 procedure that contains a loop with either a **gang** or **worker** clause. A loop in this procedure with
2376 an **auto** clause may be selected by the compiler to execute in **vector** mode, but not **worker**
2377 mode. A call to this procedure must appear in code that is executed in *vector-single* mode, though
2378 it may be in *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned*
2379 mode. For instance, a procedure with a **routine vector** directive may be called from within
2380 a loop that has the **gang** clause or the **worker** clause, but not from within a loop that has the
2381 **vector** clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may appear for each
2382 device type.

2383 **seq clause**

2384 The **seq** clause specifies that the procedure does not contain nor does it call another procedure that
2385 contains a loop with a **gang**, **worker**, or **vector** clause. A loop in this procedure with an **auto**
2386 clause will be executed in **seq** mode. A call to this procedure may appear in any mode. Only one
2387 of the **gang**, **worker**, **vector** and **seq** clauses may appear for each device type.

2388 **bind clause**

2389 The **bind** clause specifies the name to use when calling the procedure on a device other than the
2390 host. If the name is specified as an identifier, it is called as if that name were specified in the
2391 language being compiled. If the name is specified as a string, the string is used for the procedure
2392 name unmodified. A **bind** clause on a procedure definition behaves as if it had appeared on a
2393 declaration by changing the name used to call the function on a device other than the host; however,
2394 the procedure is not compiled for the device with either the original name or the name in the **bind**
2395 clause.

2396 If there is both a Fortran bind and an acc **bind** clause for a procedure definition then a call on the
2397 host will call the Fortran bound name and a call on another device will call the name in the **bind**
2398 clause.

2399 **device_type clause**

2400 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2401 **nohost** clause

2402 The **nohost** tells the compiler not to compile a version of this procedure for the host. All calls
2403 to this procedure must appear within compute regions. If this procedure is called from other pro-
2404 cedures, those other procedures must also have a matching **routine** directive with the **nohost**
2405 clause.

2406 **Restrictions**

- 2407 • Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device_type**
2408 clause.
- 2409 • At least one of the (**gang**, **worker**, **vector**, or **seq**) clauses must appear on the construct.
2410 If the **device_type** clause appears on the **routine** directive, a default level of parallelism
2411 clause must appear before the **device_type** clause, or a level of parallelism clause must
2412 appear following each **device_type** clause on the directive.
- 2413 • In C and C++, function static variables are not supported in functions to which a **routine**
2414 directive applies.
- 2415 • In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported
2416 in subprograms to which a **routine** directive applies.
- 2417 • A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- 2418 • If a function or subroutine has a **bind** clause on both the declaration and the definition then
2419 they both must bind to the same name.

2420 **2.15.2. Global Data Access**

2421 C or C++ global, file static, or *extern* variables or array, and Fortran *module* or *common block* vari-
2422 ables or arrays, that are used in accelerator routines must appear in a declare directive in a **create**,
2423 **copyin**, **device_resident** or **link** clause. If the data appears in a **device_resident**
2424 clause, the **routine** directive for the procedure must include the **nohost** clause. If the data ap-
2425 pears in a **link** clause, that data must have an active accelerator data lifetime by virtue of appearing
2426 in a data clause for a **data** construct, compute construct, or **enter data** directive.

2427 **2.16. Asynchronous Behavior**

2428 This section describes the **async** clause and the behavior of programs that use asynchronous data
2429 movement and compute constructs, and asynchronous API routines.

2430 **2.16.1. async** clause

2431 The **async** clause may appear on a **parallel**, **kernels**, or **serial** construct, or an **enter**
2432 **data**, **exit data**, **update**, or **wait** directive. In all cases, the **async** clause is optional. When
2433 there is no **async** clause on a compute or data construct, the local thread will wait until the compute
2434 construct or data operations for the current device are complete before executing any of the code

2435 that follows. When there is no **async** clause on a **wait** directive, the local thread will wait until
2436 all operations on the appropriate asynchronous activity queues for the current device are complete.
2437 When there is an **async** clause, the parallel, kernels, or serial region or data operations may be
2438 processed asynchronously while the local thread continues with the code following the construct or
2439 directive.

2440 The **async** clause may have a single *async-argument*, where an *async-argument* is a nonnegative
2441 scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special values defined
2442 below. The behavior with a negative *async-argument*, except the special values defined below, is
2443 implementation-defined. The value of the *async-argument* may be used in a **wait** directive, **wait**
2444 clause, or various runtime routines to test or wait for completion of the operation.

2445 Two special values for *async-argument* are defined in the C and Fortran header files and the Fortran
2446 **openacc** module. These are negative values, so as not to conflict with a user-specified nonnegative
2447 *async-argument*. An **async** clause with the *async-argument* **acc_async_noval** will behave
2448 the same as if the **async** clause had no argument. An **async** clause with the *async-argument*
2449 **acc_async_sync** will behave the same as if no **async** clause appeared.

2450 The *async-value* of any operation is the value of the *async-argument*, if it appears, or the value
2451 of *acc-default-async-var* if it is **acc_async_noval** or if the **async** clause had no value, or
2452 **acc_async_sync** if no **async** clause appeared. If the current device supports asynchronous
2453 operation with one or more device activity queues, the *async-value* is used to select the queue on
2454 the current device onto which to enqueue an operation. The properties of the current device and the
2455 implementation will determine how many actual activity queues are supported, and how the *async-*
2456 *value* is mapped onto the actual activity queues. Two asynchronous operations with the same current
2457 device and the same *async-value* will be enqueued onto the same activity queue, and therefore will
2458 be executed on the device in the order they are encountered by the local thread. Two asynchronous
2459 operations with different *async-values* may be enqueued onto different activity queues, and therefore
2460 may be executed on the device in either order relative to each other. If there are two or more host
2461 threads executing and sharing the same device, two asynchronous operations with the same *async-*
2462 *value* will be enqueued on the same activity queue. If the threads are not synchronized with respect
2463 to each other, the operations may be enqueued in either order and therefore may execute on the
2464 device in either order. Asynchronous operations enqueued to different devices may execute in any
2465 order, regardless of the *async-value* used for each.

2466 2.16.2. wait clause

2467 The **wait** clause may appear on a **parallel**, **kernels**, or **serial** construct, or an **enter**
2468 **data**, **exit data**, or **update** directive. In all cases, the **wait** clause is optional. When there
2469 is no **wait** clause, the associated compute or update operations may be enqueued or launched or
2470 executed immediately on the device. If there is an argument to the **wait** clause, it must be a *wait-*
2471 *argument* (See 2.16.3). The compute, data, or update operation may not be launched or executed
2472 until all operations enqueued up to this point by this thread on the associated asynchronous device
2473 activity queues have completed. One legal implementation is for the local thread to wait for all
2474 the associated asynchronous device activity queues. Another legal implementation is for the local
2475 thread to enqueue the compute, data, or update operation in such a way that the operation will
2476 not start until the operations enqueued on the associated asynchronous device activity queues have
2477 completed.

2478 **2.16.3. Wait Directive**

2479 **Summary** The **wait** directive causes the local thread or a device activity queue on the current
 2480 device to wait for completion of asynchronous operations, such as an accelerator parallel, kernels,
 2481 or serial region or an **update** directive.

2482 **Syntax** In C and C++, the syntax of the **wait** directive is:

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

2483 In Fortran the syntax of the **wait** directive is:

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

2484 where *clause* is:

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

2485 The wait argument, if it appears, must be a *wait-argument* where *wait-argument* is:

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

2486 If there is no wait argument and no **async** clause, the local thread will wait until all operations
 2487 enqueued by this thread on any activity queue on the current device have completed.

2488 If there are one or more *int-expr* expressions and no **async** clause, the local thread will wait
 2489 until all operations enqueued by this thread on each of the associated device activity queues have
 2490 completed. If a **devnum** modifier exists in the *wait-argument* then the device activity queues in the
 2491 *int-expr* expressions apply to the queues on that device number of the current device type. If no
 2492 **devnum** modifier exists then the expressions apply to the current device. It is an error to specify a
 2493 device number that is not between 0 and the number of available devices of the current device type
 2494 minus 1.

2495 The **queues** modifier within a *wait-argument* is optional to improve clarity of the expression list.

2496 If there are two or more threads executing and sharing the same device, a **wait** directive with no
 2497 **async** clause will cause the local thread to wait until all of the appropriate asynchronous opera-
 2498 tions previously enqueued by that thread have completed. To guarantee that operations have been
 2499 enqueued by other threads requires additional synchronization between those threads. There is no
 2500 guarantee that all the similar asynchronous operations initiated by other threads will have completed.

2501 If there is an **async** clause, no new operation may be launched or executed on the **async** activ-
 2502 ity queue on the current device until all operations enqueued up to this point by this thread on the
 2503 asynchronous activity queues associated with the wait argument have completed. One legal imple-
 2504 mentation is for the local thread to wait for all the associated asynchronous device activity queues.

2505 Another legal implementation is for the thread to enqueue a synchronization operation in such a
 2506 way that no new operation will start until the operations enqueued on the associated asynchronous
 2507 device activity queues have completed.

2508 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 2509 perform the wait operation unconditionally. When an **if** clause appears, the implementation will
 2510 generate code to conditionally perform the wait operation only when the *condition* evaluates to
 2511 nonzero in C or C++, or **.true.** in Fortran.

2512 A **wait** directive is functionally equivalent to a call to one of the **acc_wait**, **acc_wait_async**,
 2513 **acc_wait_all** or **acc_wait_all_async** runtime API routines, as described in Sections 3.2.13,
 2514 3.2.15, 3.2.17 and 3.2.19.

2515 Restrictions

- 2516 • The *int-expr* that appears in a **devnum** modifier must be a legal device number of the current
 2517 device type.

2518 2.17. Fortran Optional Arguments

2519 This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function
 2520 **PRESENT(arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument
 2521 for **arg** was present in the argument list of the call site. This should not be confused with the
 2522 OpenACC **present** data clause.

2523 The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no
 2524 effect at runtime if **PRESENT(arg)** is **.false.**:

- 2525 • in data clauses on compute and **data** constructs;
- 2526 • in data clauses on **enter data** and **exit data** directives;
- 2527 • in data and **device_resident** clauses on **declare** directives;
- 2528 • in **use_device** clauses on **host_data** directives;
- 2529 • in **self**, **host**, and **device** clauses on **update** directives.

2530 The appearance of a Fortran optional argument **arg** in the following situations may result in unde-
 2531 fined behavior if **PRESENT(arg)** is **.false.** when the associated construct is executed:

- 2532 • as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- 2533 • as a *var* in **cache** directives;
- 2534 • as part of an expression in any clause or directive.

2535 A call to the Fortran intrinsic function **PRESENT** behaves the same way in a compute construct or
 2536 an accelerator routine as on the host. The function call **PRESENT(arg)** must return the same value
 2537 in a compute construct as **PRESENT(arg)** would outside of the compute construct. If a Fortran
 2538 optional argument **arg** appears as an actual argument in a procedure call in a compute construct
 2539 or an accelerator routine, and the associated dummy argument **subarg** also has the **optional**
 2540 attribute, then **PRESENT(subarg)** returns the same value as **PRESENT(subarg)** would when
 2541 executed on the host.

3. Runtime Library

2543 This chapter describes the OpenACC runtime library routines that are available for use by program-
2544 mers. Use of these routines may limit portability to systems that do not support the OpenACC API.
2545 Conditional compilation using the `_OPENACC` preprocessor variable may preserve portability.

2546 This chapter has two sections:

- 2547 • Runtime library definitions
- 2548 • Runtime library routines

2549 There are four categories of runtime routines:

- 2550 • Device management routines, to get the number of devices, set the current device, and so on.
- 2551 • Asynchronous queue management, to synchronize until all activities on an async queue are
2552 complete, for instance.
- 2553 • Device test routine, to test whether this statement is executing on the device or not.
- 2554 • Data and memory management, to manage memory allocation or copy data between memo-
2555 ries.

3.1. Runtime Library Definitions

2557 In C and C++, prototypes for the runtime library routines described in this chapter are provided in
2558 a header file named `openacc.h`. All the library routines are *extern* functions with “C” linkage.
2559 This file defines:

- 2560 • The prototypes of all routines in the chapter.
- 2561 • Any datatypes used in those prototypes, including an enumeration type to describe the sup-
2562 ported device types.
- 2563 • The values of `acc_async_noval`, `acc_async_sync`, and `acc_async_default`.

2564 In Fortran, interface declarations are provided in a Fortran module named `openacc`. The `openacc`
2565 module defines:

- 2566 • The integer parameter `openacc_version` with a value `yyyymm` where `yyyy` and `mm` are the
2567 year and month designations of the version of the Accelerator programming model supported.
2568 This value matches the value of the preprocessor variable `_OPENACC`.
- 2569 • Interfaces for all routines in the chapter.
- 2570 • Integer parameters to define integer kinds for arguments to and return values for those rou-
2571 tines.

- 2572 • Integer parameters to describe the supported device types.
- 2573 • Integer parameters to define the values of `acc_async_noval`, `acc_async_sync`, and
- 2574 `acc_async_default`.

2575 Many of the routines accept or return a value corresponding to the type of device. In C and C++, the
 2576 datatype used for device type values is `acc_device_t`; in Fortran, the corresponding datatype
 2577 is `integer(kind=acc_device_kind)`. The possible values for device type are implemen-
 2578 tation specific, and are defined in the C or C++ include file `openacc.h` and the Fortran module
 2579 `openacc`. Four values are always supported: `acc_device_none`, `acc_device_default`,
 2580 `acc_device_host` and `acc_device_not_host`. For other values, look at the appropriate
 2581 files included with the implementation, or read the documentation for the implementation. The
 2582 value `acc_device_default` will never be returned by any function; its use as an argument will
 2583 tell the runtime library to use the default device type for that implementation.

2584 3.2. Runtime Library Routines

2585 In this section, for the C and C++ prototypes, pointers are typed `h_void*` or `d_void*` to desig-
 2586 nate a host memory address or device memory address, when these calls are executed on the host,
 2587 as if the following definitions were included:

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

2588 Except for `acc_on_device`, these routines are only available on the host.

2589 3.2.1. `acc_get_num_devices`

2590 **Summary** The `acc_get_num_devices` routine returns the number of devices of the given
 2591 type available.

2592 **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
```

2593 **Description** The `acc_get_num_devices` routine returns the number of devices of the given
 2594 type available. The argument tells what kind of device to count.

2595 **Restrictions**

- 2596 • This routine may not be called within a compute region.

2597 3.2.2. `acc_set_device_type`

2598 **Summary** The `acc_set_device_type` routine tells the runtime which type of device to use
2599 when executing a compute region and sets the value of `acc-current-device-type-var`. This is useful
2600 when the implementation allows the program to be compiled to use more than one type of device.

2601 **Format**

C or C++:

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

Fortran:

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

2602 **Description** The `acc_set_device_type` routine tells the runtime which type of device to
2603 use among those available and sets the value of `acc-current-device-type-var` for the current thread.
2604 A call to `acc_set_device_type` is functionally equivalent to a `set device_type` directive
2605 with the matching device type argument, as described in Section 2.14.3.

2606 **Restrictions**

- 2607 • If the device type specified is not available, the behavior is implementation-defined; in partic-
2608 ular, the program may abort.
- 2609 • If some compute regions are compiled to only use one device type, calling this routine with a
2610 different device type may produce undefined behavior.

2611 3.2.3. `acc_get_device_type`

2612 **Summary** The `acc_get_device_type` routine returns the value of `acc-current-device-type-`
2613 `var`, which is the device type of the current device. This is useful when the implementation allows
2614 the program to be compiled to use more than one type of device.

2615 **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
```

2616 **Description** The `acc_get_device_type` routine returns the value of *acc-current-device-*
 2617 *type-var* for the current thread to tell the program what type of device will be used to run the next
 2618 compute region, if one has been selected. The device type may have been selected by the program
 2619 with an `acc_set_device_type` call, with an environment variable, or by the default behavior
 2620 of the program.

2621 **Restrictions**

- 2622 • If the device type has not yet been selected, the value `acc_device_none` may be returned.

2623 **3.2.4. acc_set_device_num**

2624 **Summary** The `acc_set_device_num` routine tells the runtime which device to use and sets
 2625 the value of *acc-current-device-num-var*.

2626 **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
```

2627 **Description** The `acc_set_device_num` routine tells the runtime which device to use among
 2628 those available of the given type for compute or data regions in the current thread and sets the value
 2629 of *acc-current-device-num-var*. If the value of `devicenum` is negative, the runtime will revert to
 2630 its default behavior, which is implementation-defined. If the value of the second argument is zero,
 2631 the selected device number will be used for all device types. A call to `acc_set_device_num`
 2632 is functionally equivalent to a `set device_num` directive with the matching device number argu-
 2633 ment, as described in Section 2.14.3.

2634 **Restrictions**

- 2635 • If the value of `devicenum` is greater than or equal to the value returned by `acc_get_num_devices`
 2636 for that device type, the behavior is implementation-defined.
- 2637 • Calling `acc_set_device_num` implies a call to `acc_set_device_type` with that
 2638 device type argument.

2639 **3.2.5. acc_get_device_num**

2640 **Summary** The `acc_get_device_num` routine returns the value of *acc-current-device-num-*
 2641 *var* for the current thread.

2642 **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
```

2643 **Description** The `acc_get_device_num` routine returns the value of *acc-current-device-num-*
2644 *var* for the current thread.

2645 **3.2.6. acc_get_property**

2646 **Summary** The `acc_get_property` and `acc_get_property_string` routines return
2647 the value of a *device-property* for the specified device.

2648 **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
```

2649 **Description** The `acc_get_property` and `acc_get_property_string` routines returns
2650 the value of the specified *property*. `devicenum` and `devicetype` specify the device being
2651 queried. If `devicetype` has the value `acc_device_current`, then `devicenum` is ignored
2652 and the value of the property for the current device is returned. `property` is an enumeration
2653 constant, defined in `openacc.h`, for C or C++, or an integer parameter, defined in the `openacc`
2654 module, for Fortran. Integer-valued properties are returned by `acc_get_property`, and string-
2655 valued properties are returned by `acc_get_property_string`. In Fortran, `acc_get_property_string`
2656 returns the result into the `character` variable passed as the last argument.

2657 The supported values of `property` are given in the following table.

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

2658 An implementation may support additional properties for some devices.

2660 Restrictions

- 2661 • These routines may not be called within an compute region.
- 2662 • If the value of `property` is not one of the known values for that query routine, or that
2663 property has no value for the specified device, `acc_get_property` will return 0 and
2664 `acc_get_property_string` will return NULL (in C or C++) or an blank string (in
2665 Fortran).

2666 3.2.7. acc_init

2667 **Summary** The `acc_init` routine tells the runtime to initialize the runtime for that device type.
2668 This can be used to isolate any initialization cost from the computational cost, when collecting
2669 performance statistics.

2670 Format

C or C++:

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

Fortran:

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

2671 **Description** The `acc_init` routine also implicitly calls `acc_set_device_type`. A call to
2672 `acc_init` is functionally equivalent to a `init` directive with the matching device type argument,
2673 as described in Section 2.14.1.

2674 Restrictions

- 2675 • This routine may not be called within a compute region.
- 2676 • If the device type specified is not available, the behavior is implementation-defined; in partic-
2677 ular, the program may abort.

- 2678
- If the routine is called more than once without an intervening **acc_shutdown** call, with a different value for the device type argument, the behavior is implementation-defined.
- 2679
- 2680
- If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.
- 2681

2682 3.2.8. **acc_shutdown**

2683 **Summary** The **acc_shutdown** routine tells the runtime to shut down any connection to de-
2684 vices of the given device type, and free up any runtime resources. A call to **acc_shutdown**
2685 is functionally equivalent to a **shutdown** directive with the matching device type argument, as
2686 described in Section 2.14.2.

2687 **Format**

C or C++:

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

Fortran:

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

2688 **Description** The **acc_shutdown** routine disconnects the program from any device of the spec-
2689 ified device type. Any data that is present in the memory of any such device is immediately deallo-
2690 cated.

2691 **Restrictions**

- 2692
- This routine may not be called during execution of a compute region.
- 2693
- If the program attempts to execute a compute region on a device or to access any data in the memory of a device after a call to **acc_shutdown** for that device type, the behavior is undefined.
- 2694
- 2695
- If the program attempts to shut down the **acc_device_host** device type, the behavior is undefined.
- 2696
- 2697

2698 3.2.9. **acc_async_test**

2699 **Summary** The **acc_async_test** routine tests for completion of all associated asynchronous
2700 operations on the current device.

2701 **Format**

C or C++:

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

Fortran:

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

2702 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
 2703 If that value did not appear in any **async** clauses, or if it did appear in one or more **async** clauses
 2704 and all such asynchronous operations have completed on the current device, the **acc_async_test**
 2705 routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some such asyn-
 2706 chronous operations have not completed, the **acc_async_test** routine will return with a zero
 2707 value in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the
 2708 **acc_async_test** routine will return with a nonzero value or **.true.** only if all matching asyn-
 2709 chronous operations initiated by this thread have completed; there is no guarantee that all matching
 2710 asynchronous operations initiated by other threads have completed.

2711 3.2.10. acc_async_test_device

2712 **Summary** The **acc_async_test_device** routine tests for completion of all associated asyn-
 2713 chronous operations on a device.

2714 Format

C or C++:

```
int acc_async_test_device( int, int );
```

Fortran:

```
logical function acc_async_test_device( arg, device )
integer(acc_handle_kind) :: arg
integer :: device
```

2715 **Description** The first argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
 2716 The second argument must be a valid device number of the current device type.

2717 If the *async-argument* did not appear in any **async** clauses, or if it did appear in one or more
 2718 **async** clauses and all such asynchronous operations have completed on the specified device, the
 2719 **acc_async_test_device** routine will return with a nonzero value in C and C++, or **.true.**
 2720 in Fortran. If some such asynchronous operations have not completed, the **acc_async_test_device**
 2721 routine will return with a zero value in C and C++, or **.false.** in Fortran. If two or more threads
 2722 share the same accelerator, the **acc_async_test_device** routine will return with a nonzero
 2723 value or **.true.** only if all matching asynchronous operations initiated by this thread have com-
 2724 pleted; there is no guarantee that all matching asynchronous operations initiated by other threads
 2725 have completed.

2726 3.2.11. acc_async_test_all

2727 **Summary** The **acc_async_test_all** routine tests for completion of all asynchronous op-
 2728 erations.

2729 **Format**

C or C++:

```
int acc_async_test_all( );
```

Fortran:

```
logical function acc_async_test_all( )
```

2730 **Description** If all outstanding asynchronous operations have completed, the `acc_async_test_all`
2731 routine will return with a nonzero value in C and C++, or `.true.` in Fortran. If some asynchronous
2732 operations have not completed, the `acc_async_test_all` routine will return with a zero value
2733 in C and C++, or `.false.` in Fortran. If two or more threads share the same accelerator, the
2734 `acc_async_test_all` routine will return with a nonzero value or `.true.` only if all outstand-
2735 ing asynchronous operations initiated by this thread have completed; there is no guarantee that all
2736 asynchronous operations initiated by other threads have completed.

2737 **3.2.12. acc_async_test_all_device**

2738 **Summary** The `acc_async_test_all_device` routine tests for completion of all asyn-
2739 chronous operations.

2740 **Format**

C or C++:

```
int acc_async_test_all_device( int );
```

Fortran:

```
logical function acc_async_test_all_device( device )  
integer :: device
```

2741 **Description** The argument must be a valid device number of the current device type. If all out-
2742 standing asynchronous operations have completed on the specified device, the `acc_async_test_all_device`
2743 routine will return with a nonzero value in C and C++, or `.true.` in Fortran. If some asynchronous
2744 operations have not completed, the `acc_async_test_all_device` routine will return with a
2745 zero value in C and C++, or `.false.` in Fortran. If two or more threads share the same acceler-
2746 ator, the `acc_async_test_all_device` routine will return with a nonzero value or `.true.`
2747 only if all outstanding asynchronous operations initiated by this thread have completed; there is no
2748 guarantee that all asynchronous operations initiated by other threads have completed.

2749 **3.2.13. acc_wait**

2750 **Summary** The `acc_wait` routine waits for completion of all associated asynchronous opera-
2751 tions on the current device.

2752 **Format**

C or C++:

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

Fortran:

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

2753 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
2754 If that value appeared in one or more **async** clauses, the **acc_wait** routine will not return until
2755 the latest such asynchronous operation has completed on the current device. If two or more threads
2756 share the same accelerator, the **acc_wait** routine will return only if all matching asynchronous
2757 operations initiated by this thread have completed; there is no guarantee that all matching asyn-
2758 chronous operations initiated by other threads have completed. For compatibility with version 1.0,
2759 this routine may also be spelled **acc_async_wait**. A call to **acc_wait** is functionally equiv-
2760 alent to a **wait** directive with a matching wait argument and no **async** clause, as described in
2761 Section 2.16.3.

2762 **3.2.14. acc_wait_device**

2763 **Summary** The **acc_wait_device** routine waits for completion of all associated asynchronous
2764 operations on a device.

2765 **Format**

C or C++:

```
void acc_wait_device( int, int );
```

Fortran:

```
subroutine acc_wait_device( arg, device )  
integer(acc_handle_kind) :: arg  
integer :: device
```

2766 **Description** The first argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
2767 The second argument must be a valid device number of the current device type.

2768 If the *async-argument* appeared in one or more **async** clauses, the **acc_wait** routine will not
2769 return until the latest such asynchronous operation has completed on the specified device. If two
2770 or more threads share the same accelerator, the **acc_wait** routine will return only if all match-
2771 ing asynchronous operations initiated by this thread have completed; there is no guarantee that all
2772 matching asynchronous operations initiated by other threads have completed.

2773 **3.2.15. acc_wait_async**

2774 **Summary** The `acc_wait_async` routine enqueues a wait operation on one async queue of
2775 the current device for the operations previously enqueued on another async queue.

2776 **Format**

C or C++:

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

Fortran:

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

2777 **Description** The arguments must be *async-arguments*, as defined in Section 2.16.1 *async* clause.
2778 The routine will enqueue a wait operation on the appropriate device queue associated with the
2779 second argument, which will wait for operations enqueued on the device queue associated with
2780 the first argument. See Section 2.16 Asynchronous Behavior for more information. A call to
2781 `acc_wait_async` is functionally equivalent to a `wait` directive with a matching wait argument
2782 and a matching `async` argument, as described in Section 2.16.3.

2783 **3.2.16. acc_wait_device_async**

2784 **Summary** The `acc_wait_device_async` routine enqueues a wait operation on one async
2785 queue of a device for the operations previously enqueued on another async queue.

2786 **Format**

C or C++:

```
void acc_wait_device_async( int, int, int );
```

Fortran:

```
subroutine acc_wait_device_async( arg, async, device )  
  integer(acc_handle_kind) :: arg, async  
  integer :: device
```

2787 **Description** The first two arguments must be *async-arguments*, as defined in Section 2.16.1
2788 *async* clause. The third argument must be a valid device number of the current device type.

2789 The routine will enqueue a wait operation on the appropriate device queue associated with the
2790 second argument, which will wait for operations enqueued on the device queue associated with the
2791 first argument.

2792 See Section 2.16 Asynchronous Behavior for more information. A call to `acc_wait_device_async`
2793 is functionally equivalent to a `wait` directive with a matching wait argument and a matching `async`
2794 argument, as described in Section 2.16.3.

2795 **3.2.17. acc_wait_all**2796 **Summary** The `acc_wait_all` routine waits for completion of all asynchronous operations.2797 **Format**

C or C++:

```
void acc_wait_all( );
```

Fortran:

```
subroutine acc_wait_all( )
```

2798 **Description** The `acc_wait_all` routine will not return until the all asynchronous operations
2799 have completed. If two or more threads share the same accelerator, the `acc_wait_all` routine
2800 will return only if all asynchronous operations initiated by this thread have completed; there is no
2801 guarantee that all asynchronous operations initiated by other threads have completed. For com-
2802 patibility with version 1.0, this routine may also be spelled `acc_async_wait_all`. A call to
2803 `acc_wait_all` is functionally equivalent to a `wait` directive with no wait argument list and no
2804 `async` argument, as described in Section 2.16.3.

2805 **3.2.18. acc_wait_all_device**2806 **Summary** The `acc_wait_all_device` routine waits for completion of all asynchronous
2807 operations the specified device.2808 **Format**

C or C++:

```
void acc_wait_all_device( int );
```

Fortran:

```
subroutine acc_wait_all_device( device )  
integer :: device
```

2809 **Description** The argument must be a valid device number of the current device type. The
2810 `acc_wait_all_device` routine will not return until the all asynchronous operations have com-
2811 pleted on the specified device. If two or more threads share the same accelerator, the `acc_wait_all_device`
2812 routine will return only if all asynchronous operations initiated by this thread have completed; there
2813 is no guarantee that all asynchronous operations initiated by other threads have completed.

2814 **3.2.19. acc_wait_all_async**2815 **Summary** The `acc_wait_all_async` routine enqueues wait operations on one async queue
2816 for the operations previously enqueued on all other async queues.

2817 **Format**

C or C++:

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

Fortran:

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

2818 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
 2819 The routine will enqueue a wait operation on the appropriate device queue for each other device
 2820 queue. See Section 2.16 Asynchronous Behavior for more information. A call to **acc_wait_all_async**
 2821 is functionally equivalent to a **wait** directive with no wait argument list and a matching **async**
 2822 argument, as described in Section 2.16.3.

2823 **3.2.20. acc_wait_all_device_async**

2824 **Summary** The **acc_wait_all_device_async** routine enqueues wait operations on one
 2825 *async* queue for the operations previously enqueued on all other *async* queues on the specified
 2826 device.

2827 **Format**

C or C++:

```
void acc_wait_all_device_async( int, int );
```

Fortran:

```
subroutine acc_wait_all_device_async( async, device )
  integer(acc_handle_kind) :: async
  integer :: device
```

2828 **Description** The first argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
 2829 The second argument must be a valid device number of the current device type.
 2830 The routine will enqueue a wait operation on the appropriate device queue for each other device
 2831 queue. See Section 2.16 Asynchronous Behavior for more information. A call to **acc_wait_all_async**
 2832 is functionally equivalent to a **wait** directive with no wait argument list and a matching **async**
 2833 argument, as described in Section 2.16.3.

2834 **3.2.21. acc_get_default_async**

2835 **Summary** The **acc_get_default_async** routine returns the value of *acc-default-async-*
 2836 *var* for the current thread.

2837 **Format**

C or C++:

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

Fortran:

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

2838 **Description** The `acc_get_default_async` routine returns the value of `acc-default-async-`
2839 `var` for the current thread, which is the asynchronous queue used when an `async` clause appears
2840 without an `async-argument` or with the value `acc_async_noval`.

2841 **3.2.22. acc_set_default_async**

2842 **Summary** The `acc_set_default_async` routine tells the runtime which asynchronous queue
2843 to use when an `async` clause appears with no queue argument.

2844 **Format**

C or C++:

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

Fortran:

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

2845 **Description** The `acc_set_default_async` routine tells the runtime to place any directives
2846 with an `async` clause that does not have an `async-argument` or with the special `acc_async_noval`
2847 value into the specified asynchronous activity queue instead of the default asynchronous activity
2848 queue for that device by setting the value of `acc-default-async-var` for the current thread. The spe-
2849 cial argument `acc_async_default` will reset the default asynchronous activity queue to the
2850 initial value, which is implementation-defined. A call to `acc_set_default_async` is func-
2851 tionally equivalent to a `set default_async` directive with a matching argument in `int-expr`, as
2852 described in Section 2.14.3.

2853 **3.2.23. acc_on_device**

2854 **Summary** The `acc_on_device` routine tells the program whether it is executing on a partic-
2855 ular device.

2856 **Format**

C or C++:

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

Fortran:

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

2857 **Description** The `acc_on_device` routine may be used to execute different paths depend-
2858 ing on whether the code is running on the host or on some accelerator. If the `acc_on_device`
2859 routine has a compile-time constant argument, it evaluates at compile time to a constant. The ar-
2860 gument must be one of the defined accelerator types. If the argument is `acc_device_host`,
2861 then outside of a compute region or accelerator routine, or in a compute region or accelerator rou-
2862 tine that is executed on the host CPU, this routine will evaluate to nonzero for C or C++, and
2863 `.true.` for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.` for Fortran.
2864 If the argument is `acc_device_not_host`, the result is the negation of the result with argu-
2865 ment `acc_device_host`. If the argument is an accelerator device type, then in a compute region
2866 or routine that is executed on a device of that type, this routine will evaluate to nonzero for C or
2867 C++, and `.true.` for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.` for
2868 Fortran. The result with argument `acc_device_default` is undefined.

2869 **3.2.24. acc_malloc**

2870 **Summary** The `acc_malloc` routine allocates space in the current device memory.

2871 **Format**

C or C++:

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

2872 **Description** The `acc_malloc` routine may be used to allocate space in the current device
2873 memory. Pointers assigned from this function may be used in `deviceptr` clauses to tell the
2874 compiler that the pointer target is resident on the device. In case of an error, `acc_malloc` returns
2875 a NULL pointer.

2876 **3.2.25. acc_free**

2877 **Summary** The `acc_free` routine frees memory on the current device.

2878 **Format**

C or C++:

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

2879 **Description** The `acc_free` routine will free previously allocated space in the current device
 2880 memory; the argument should be a pointer value that was returned by a call to `acc_malloc`. If
 2881 the argument is a NULL pointer, no operation is performed.

2882 3.2.26. `acc_copyin`

2883 **Summary** The `acc_copyin` routines test to see if the argument is in shared memory or already
 2884 present in the current device memory; if not, they allocate space in the current device memory to
 2885 correspond to the specified local memory, and copy the data to that device memory.

2886 **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
```

2887 **Description** The `acc_copyin` routines are equivalent to the `enter data` directive with a
 2888 `copyin` clause, as described in Section 2.7.6. In C, the arguments are a pointer to the data and
 2889 length in bytes; the synchronous function returns a pointer to the allocated device memory, as with
 2890 `acc_malloc`. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 2891 section of intrinsic type. In the second, the first argument is a variable or array element and the
 2892 second is the length in bytes.

2893 The behavior of the `acc_copyin` routines is:

- 2894 • If the data is in shared memory, no action is taken. The C `acc_copyin` returns the incoming
 2895 pointer.
- 2896 • If the data is present in the current device memory, a *present increment* action with the dy-
 2897 namic reference counter is performed. The C `acc_copyin` returns a pointer to the existing
 2898 device memory.
- 2899 • Otherwise, a *copyin* action with the dynamic reference counter is performed. The C `acc_copyin`
 2900 returns the device address of the newly allocated memory.

2901 This data may be accessed using the `present` data clause. Pointers assigned from the C `acc_copyin`
 2902 function may be used in `deviceptr` clauses to tell the compiler that the pointer target is resident
 2903 on the device.

2904 The **_async** versions of this function will perform any data transfers asynchronously on the async
 2905 queue associated with the value passed in as the **async** argument. The function may return be-
 2906 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
 2907 synchronous versions will not return until the data has been completely transferred.

2908 For compatibility with OpenACC 2.0, **acc_present_or_copyin** and **acc_pcopyin** are al-
 2909 ternate names for **acc_copyin**.

2910 3.2.27. acc.create

2911 **Summary** The **acc_create** routines test to see if the argument is in shared memory or already
 2912 present in the current device memory; if not, they allocate space in the current device memory to
 2913 correspond to the specified local memory.

2914 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
```

2915 **Description** The **acc_create** routines are equivalent to the **enter data** directive with a
 2916 **create** clause, as described in Section 2.7.8. In C, the arguments are a pointer to the data and
 2917 length in bytes; the synchronous function returns a pointer to the allocated device memory, as with
 2918 **acc_malloc**. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 2919 section of intrinsic type. In the second, the first argument is a variable or array element and the
 2920 second is the length in bytes.

2921 The behavior of the **acc_create** routines is:

- 2922 • If the data is in shared memory, no action is taken. The C **acc_create** returns the incoming
 2923 pointer.
- 2924 • If the data is present in the current device memory, a *present increment* action with the dy-
 2925 namic reference counter is performed. The C **acc_create** returns a pointer to the existing
 2926 device memory.
- 2927 • Otherwise, a *create* action with the dynamic reference counter is performed. The C **acc_create**
 2928 returns the device address of the newly allocated memory.

2929 This data may be accessed using the **present** data clause. Pointers assigned from the C **acc_copyin**
 2930 function may be used in **deviceptr** clauses to tell the compiler that the pointer target is resident
 2931 on the device.

2932 The **_async** versions of these function may perform the data allocation asynchronously on the
 2933 async queue associated with the value passed in as the **async** argument. The synchronous versions
 2934 will not return until the data has been allocated.

2935 For compatibility with OpenACC 2.0, **acc_present_or_create** and **acc_pcreate** are al-
 2936 ternate names for **acc_create**.

2937 3.2.28. acc_copyout

2938 **Summary** The **acc_copyout** routines test to see if the argument is in shared memory; if not,
 2939 the argument must be present in the current device memory, and the routines copy data from device
 2940 memory to the corresponding local memory, then deallocate that space from the device memory.

2941 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
```

2942 **Description** The **acc_copyout** routines are equivalent to the **exit data** directive with a
 2943 **copyout** clause, and the **acc_copyout_finalize** routines are equivalent to the **exit data**
 2944 directive with both **copyout** and **finalize** clauses, as described in Section 2.7.7. In C, the
 2945 arguments are a pointer to the data and length in bytes. In Fortran, two forms are supported. In the
 2946 first, the argument is a contiguous array section of intrinsic type. In the second, the first argument
 2947 is a variable or array element and the second is the length in bytes.

2948 The behavior of the **acc_copyout** routines is:

- 2949 • If the data is in shared memory, no action is taken.

- 2950 • Otherwise, if the data is not present in the current device memory, a runtime error is issued.
- 2951 • Otherwise, a *present decrement* action with the dynamic reference counter is performed (**acc_copyout**),
- 2952 or the dynamic reference counter is set to zero (**acc_copyout_finalize**). If both ref-
- 2953 erence counters are then zero, a *copyout* action is performed.

2954 The **_async** versions of these functions will perform any associated data transfers asynchronously

2955 on the async queue associated with the value passed in as the **async** argument. The function may

2956 return before the data has been transferred or deallocated; see Section 2.16 Asynchronous Behavior

2957 for more details. The synchronous versions will not return until the data has been completely trans-

2958 ferred. Even if the data has not been transferred or deallocated before the function returns, the data

2959 will be treated as not present in the current device memory.

2960 3.2.29. acc_delete

2961 **Summary** The **acc_delete** routines test to see if the argument is in shared memory; if not,

2962 the argument must be present in the current device memory, and the routines deallocate that space

2963 from the device memory.

2964 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
```

2965 **Description** The **acc_delete** routines are equivalent to the **exit data** directive with a

2966 **delete** clause,

2967 and the **acc_delete_finalize** routines are equivalent to the **exit data** directive with both

2968 **delete** clause and **finalize** clauses, as described in Section 2.7.10. The arguments are as for

2969 **acc_copyout**.

2970 The behavior of the **acc_delete** routines is:

- 2971 • If the data is in shared memory, no action is taken.
- 2972 • Otherwise, if the data is not present in the current device memory, a runtime error is issued.
- 2973 • Otherwise, a *present decrement* action with the dynamic reference counter is performed (**acc_delete**),
- 2974 or the dynamic reference counter is set to zero (**acc_delete_finalize**). If both refer-
- 2975 ence counters are then zero, a *delete* action is performed.

2976 The **_async** versions of these function may perform the data deallocation asynchronously on the
 2977 async queue associated with the value passed in as the **async** argument. The synchronous versions
 2978 will not return until the data has been deallocated. Even if the data has not been deallocated before
 2979 the function returns, the data will be treated as not present in the current device memory.

2980 3.2.30. acc_update_device

2981 **Summary** The **acc_update_device** routines test to see if the argument is in shared memory;
 2982 if not, the argument must be present in the current device memory, and the routines update the data
 2983 in device memory from the corresponding local memory.

2984 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_async( a, async )
subroutine acc_update_device_async( a, len, async )
  type(*), dimension(..) :: a
  integer :: len
  integer(acc_handle_kind) :: async
```

2985 **Description** The **acc_update_device** routine is equivalent to the **update** directive with a
 2986 **device** clause, as described in Section 2.14.4. In C, the arguments are a pointer to the data and
 2987 length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 2988 section of intrinsic type. In the second, the first argument is a variable or array element and the
 2989 second is the length in bytes. For data not in shared memory, the data in the local memory is copied
 2990 to the corresponding device memory. It is a runtime error to call this routine if the data is not present
 2991 in the current device memory.

2992 The **_async** versions of this function will perform the data transfers asynchronously on the async
 2993 queue associated with the value passed in as the **async** argument. The function may return be-
 2994 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
 2995 synchronous versions will not return until the data has been completely transferred.

2996 **3.2.31. acc_update_self**

2997 **Summary** The `acc_update_self` routines test to see if the argument is in shared memory;
 2998 if not, the argument must be present in the current device memory, and the routines update the data
 2999 in local memory from the corresponding device memory.

3000 **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
```

3001 **Description** The `acc_update_self` routine is equivalent to the `update` directive with a
 3002 `self` clause, as described in Section 2.14.4. In C, the arguments are a pointer to the data and
 3003 length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 3004 section of intrinsic type. In the second, the first argument is a variable or array element and the
 3005 second is the length in bytes. For data not in shared memory, the data in the local memory is copied
 3006 to the corresponding device memory. There must be a device copy of the data on the device when
 3007 calling this routine, otherwise no action is taken by the routine. It is a runtime error to call this
 3008 routine if the data is not present in the current device memory.

3009 The `_async` versions of this function will perform the data transfers asynchronously on the `async`
 3010 queue associated with the value passed in as the `async` argument. The function may return be-
 3011 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
 3012 synchronous versions will not return until the data has been completely transferred.

3013 **3.2.32. acc_map_data**

3014 **Summary** The `acc_map_data` routine maps previously allocated space in the current device
 3015 memory to the specified host data.

3016 **Format**

C or C++:

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

3017 **Description** The `acc_map_data` routine is similar to an `enter data` directive with a `create`
3018 clause, except instead of allocating new device memory to start a data lifetime, the device address
3019 to use for the data lifetime is specified as an argument. The first argument is a host address, fol-
3020 lowed by the corresponding device address and the data length in bytes. After this call, when the
3021 host data appears in a data clause, the specified device memory will be used. It is an error to call
3022 `acc_map_data` for host data that is already present in the current device memory. It is undefined
3023 to call `acc_map_data` with a device address that is already mapped to host data. The device
3024 address may be the result of a call to `acc_malloc`, or may come from some other device-specific
3025 API routine. After mapping the device memory, the dynamic reference count for the host data is set
3026 to one, but no data movement will occur. Memory mapped by `acc_map_data` may not have the
3027 associated dynamic reference count decremented to zero, except by a call to `acc_unmap_data`.
3028 See Section 2.6.7 Reference Counters.

3029 3.2.33. `acc_unmap_data`

3030 **Summary** The `acc_unmap_data` routine unmaps device data from the specified host data.

3031 **Format**

C or C++:

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

3032 **Description** The `acc_unmap_data` routine is similar to an `exit data` directive with a
3033 `delete` clause, except the device memory is not deallocated. The argument is pointer to the host
3034 data. A call to this routine ends the data lifetime for the specified host data. The device memory is
3035 not deallocated. It is undefined behavior to call `acc_unmap_data` with a host address unless that
3036 host address was mapped to device memory using `acc_map_data`. After unmapping memory the
3037 dynamic reference count for the pointer is set to zero, but no data movement will occur. It is an
3038 error to call `acc_unmap_data` if the structured reference count for the pointer is not zero. See
3039 Section 2.6.7 Reference Counters.

3040 3.2.34. `acc_deviceptr`

3041 **Summary** The `acc_deviceptr` routine returns the device pointer associated with a specific
3042 host address.

3043 **Format**

C or C++:

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

3044 **Description** The `acc_deviceptr` routine returns the device pointer associated with a host
3045 address. The argument is the address of a host variable or array that has an active lifetime on the
3046 current device. If the data is not present in the current device memory, the routine returns a NULL
3047 value.

3048 **3.2.35. acc_hostptr**

3049 **Summary** The `acc_hostptr` routine returns the host pointer associated with a specific device
3050 address.

3051 **Format**

C or C++:

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

3052 **Description** The `acc_hostptr` routine returns the host pointer associated with a device ad-
3053 dress. The argument is the address of a device variable or array, such as that returned from `acc_deviceptr`,
3054 `acc_create` or `acc_copyin`. If the device address is NULL, or does not correspond to any host
3055 address, the routine returns a NULL value.

3056 **3.2.36. acc_is_present**

3057 **Summary** The `acc_is_present` routine tests whether a variable or array region is accessible
3058 from the current device.

3059 **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
```

3060 **Description** The `acc_is_present` routine tests whether the specified host data is accessible
3061 from the current device. In C, the arguments are a pointer to the data and length in bytes; the
3062 function returns nonzero if the specified data is fully present, and zero otherwise. In Fortran, two
3063 forms are supported. In the first, the argument is a contiguous array section of intrinsic type. In the
3064 second, the first argument is a variable or array element and the second is the length in bytes. The
3065 function returns `.true.` if the specified data is in shared memory or is fully present, and `.false.`
3066 otherwise. If the byte length is zero, the function returns nonzero in C or `.true.` in Fortran if the
3067 given address is in shared memory or is present at all in the current device memory.

3068 **3.2.37. acc_memcpy_to_device**

3069 **Summary** The `acc_memcpy_to_device` routine copies data from local memory to device
3070 memory.

3071 **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 );
```

3072 **Description** The `acc_memcpy_to_device` routine copies `bytes` of data from the local
3073 address in `src` to the device address in `dest`. The destination address must be an address accessible
3074 from the current device, such as an address returned from `acc_malloc` or `acc_deviceptr`, or
3075 an address in shared memory.

3076 The `_async` version of this function will perform the data transfers asynchronously on the `async`
3077 queue associated with the value passed in as the `async` argument. The function may return be-
3078 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
3079 synchronous versions will not return until the data has been completely transferred.

3080 **3.2.38. acc_memcpy_from_device**

3081 **Summary** The `acc_memcpy_from_device` routine copies data from device memory to lo-
3082 cal memory.

3083 **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 );
```

3084 **Description** The `acc_memcpy_from_device` routine copies `bytes` data from the device
3085 address in `src` to the local address in `dest`. The source address must be an address accessible
3086 from the current device, such as an address returned from `acc_malloc` or `acc_deviceptr`,
3087 or an address in shared memory.

3088 The `_async` version of this function will perform the data transfers asynchronously on the `async`
3089 queue associated with the value passed in as the `async` argument. The function may return be-
3090 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
3091 synchronous versions will not return until the data has been completely transferred.

3092 **3.2.39. acc_memcpy_device**

3093 **Summary** The `acc_memcpy_device` routine copies data from one memory location to an-
3094 other memory location on the current device.

3095 **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 );
```

3096 **Description** The `acc_memcpy_device` routine copies `bytes` data from the device address
3097 in `src` to the device address in `dest`. Both addresses must be addresses in the current device
3098 memory, such as would be returned from `acc_malloc` or `acc_deviceptr`. If `dest` and `src`
3099 overlap, the behavior is undefined.

3100 The `_async` version of this function will perform the data transfers asynchronously on the `async`
3101 queue associated with the value passed in as the `async` argument. The function may return be-
3102 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
3103 synchronous versions will not return until the data has been completely transferred.

3104 **3.2.40. acc_attach**

3105 **Summary** The `acc_attach` routine updates a pointer in device memory to point to the corre-
3106 sponding device copy of the host pointer target.

3107 **Format**

C or C++:

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

3108 **Description** The `acc_attach` routines are passed the address of a host pointer. If the data is
3109 in shared memory, or if the pointer `*ptr` is in shared memory or is not present in the current device
3110 memory, or the address to which the `*ptr` points is not present in the current device memory, no
3111 action is taken. Otherwise, these routines perform the *attach* action (Section 2.7.2).

3112 These routines may issue a data transfer from local memory to device memory. The `_async`
3113 version of this function will perform the data transfers asynchronously on the `async` queue associated
3114 with the value passed in as the `async` argument. The function may return before the data has been
3115 transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous version
3116 will not return until the data has been completely transferred.

3117 **3.2.41. acc_detach**

3118 **Summary** The `acc_detach` routine updates a pointer in device memory to point to the host
3119 pointer target.

3120 **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 );

```

3121 **Description** The `acc_detach` routines are passed the address of a host pointer. If the data is
3122 in shared memory, or if the pointer `*ptr` is in shared memory or is not present in the current device
3123 memory, if the *attachment counter* for the pointer `*ptr` is zero, no action is taken. Otherwise, these
3124 routines perform the *detach* action (Section 2.7.2).

3125 The `acc_detach_finalize` routines are equivalent to an **exit data** directive with **detach**
3126 and **finalize** clauses, as described in Section 2.7.12 detach clause. If the data is in shared
3127 memory, or if the pointer `*ptr` is not present in the current device memory, or if the *attachment*
3128 *counter* for the pointer `*ptr` is zero, no action is taken. Otherwise, these routines perform the
3129 *immediate detach* action (Section 2.7.2).

3130 These routines may issue a data transfer from local memory to device memory. The `_async`
3131 versions of these functions will perform the data transfers asynchronously on the async queue asso-
3132 ciated with the value passed in as the **async** argument. These functions may return before the data
3133 has been transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous
3134 versions will not return until the data has been completely transferred.

3135 **3.2.42. acc_memcpy_d2d**

3136 **Summary** This `acc_memcpy_d2d` and `acc_memcpy_d2d_async` routines copy the con-
3137 tents of an array on one device to an array on the same or a different device without updating the
3138 value on the host.

3139 **Format**

C or C++:

```

void acc_memcpy_d2d( hvoid* dst, hvoid* src,
                    size_t sz, int dstdev, int srcdev);
void acc_memcpy_d2d_async( hvoid* dst, hvoid* src,
                           size_t sz, int dstdev, int srcdev,
                           int srcasync);

```

Fortran:

```

subroutine acc_memcpy_d2d( dst, src, sz, dstdev, srcdev )
subroutine acc_memcpy_d2d_async( dst, src, sz, dstdev, srcdev )
    type(*), dimension(..) :: dst
    type(*), dimension(..) :: src
    integer :: sz

```

```
integer :: dstdev
integer :: srcdev
integer :: srcasync
```

3140 **Description** The `acc_memcpy_d2d` and `acc_memcpy_d2d_async` routines are passed the
3141 address of destination and source host pointers as well as integer device numbers for the destination
3142 and source devices, which must both be of the current device type. If both arrays are in shared
3143 memory, then no action is taken. If either pointer is not in shared memory, then that array must be
3144 present on its respective device. If these conditions are met, the contents of the source array on the
3145 source device are copied to the destination array on the destination device.

3146 For `acc_memcpy_d2d_async` the value of `srcasync` is the number of an async queue on the
3147 source device. This routine will issue the copy operation into the device activity queue for the
3148 source device and follow the usual asynchronous device queue semantics defined in 2.16.

3149 4. Environment Variables

3150 This chapter describes the environment variables that modify the behavior of accelerator regions.
3151 The names of the environment variables must be upper case. The values assigned environment
3152 variables are case-insensitive and may have leading and trailing white space. If the values of the
3153 environment variables change after the program has started, even if the program itself modifies the
3154 values, the behavior is implementation-defined.

3155 4.1. ACC_DEVICE_TYPE

3156 The **ACC_DEVICE_TYPE** environment variable controls the default device type to use when ex-
3157 ecuting parallel, kernels, and serial regions, if the program has been compiled to use more than
3158 one different type of device. The allowed values of this environment variable are implementa-
3159 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
```

3160 4.2. ACC_DEVICE_NUM

3161 The **ACC_DEVICE_NUM** environment variable controls the default device number to use when
3162 executing accelerator regions. The value of this environment variable must be a nonnegative integer
3163 between zero and the number of devices of the desired type attached to the host. If the value is
3164 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
```

3165 4.3. ACC_PROFLIB

3166 The **ACC_PROFLIB** environment variable specifies the profiling library. More details about the
3167 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

3169 This chapter describes the OpenACC interface for tools that can be used for profile and trace data
3170 collection. Therefore it provides a set of OpenACC-specific event callbacks that are triggered dur-
3171 ing the application run. Currently, this interface does not support tools that employ asynchronous
3172 sampling. In this chapter, the term *runtime* refers to the OpenACC runtime library. The term *library*
3173 refers to the third party routines invoked at specified events by the OpenACC runtime.

3174 There are four steps for interfacing a *library* to the *runtime*. The first is to write the data collection
3175 library callback routines. Section 5.1 Events describes the supported runtime events and the order
3176 in which callbacks to the callback routines will occur. Section 5.2 Callbacks Signature describes
3177 the signature of the callback routines for all events.

3178 The second is to use registration routines to register the data collection callbacks for the appropriate
3179 events. The data collection and registration routines are then saved in a static or dynamic library
3180 or shared object. The third is to load the *library* at runtime. The *library* may be statically linked
3181 to the application or dynamically loaded by the application or by the *runtime*. This is described in
3182 Section 5.3 Loading the Library.

3183 The fourth step is to invoke the registration routine to register the desired callbacks with the events.
3184 This may be done explicitly by the application, if the library is statically linked with the application,
3185 implicitly by including a call to the registration routine in a `.init` section, or by including an
3186 initialization routine in the library if it is dynamically loaded by the *runtime*. This is described in
3187 Section 5.4 Registering Event Callbacks.

3188 Subsequently, the *library* may collect information when the callback routines are invoked by the
3189 *runtime* and process or store the acquired data.

5.1. Events

3191 This section describes the events that are recognized by the runtime. Most events may have a start
3192 and end callback routine, that is, a routine that is called just before the runtime code to handle
3193 the event starts and another routine that is called just after the event is handled. The event names
3194 and routine prototypes are available in the header file `acc_prof.h`, which is delivered with the
3195 OpenACC implementation. Event names are prefixed with `acc_ev_`.

3196 The ordering of events must reflect the order in which the OpenACC runtime actually executes them,
3197 i.e. if a runtime moves the enqueueing of data transfers or kernel launches outside the originating
3198 clauses/constructs, it needs to issue the corresponding launch callbacks when they really occur. A
3199 callback for a start event must always precede the matching end callback. The behavior of a tool
3200 receiving a callback after the runtime shutdown callback is undefined.

3201 The events that the runtime supports can be registered with a callback and are defined in the enu-
3202 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;
```

3203 5.1.1. Runtime Initialization and Shutdown

3204 No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is
3205 handled as described in Section 5.3 Loading the Library.

3206 The *runtime shutdown* event name is

```
acc_ev_runtime_shutdown
```

3207 The **acc_ev_runtime_shutdown** event is triggered before the OpenACC runtime shuts down,
3208 either because all devices have been shutdown by calls to the **acc_shutdown** API routine, or at
3209 the end of the program.

3210 5.1.2. Device Initialization and Shutdown

3211 The *device initialization* event names are

```
acc_ev_device_init_start
```

acc_ev_device_init_end

3212 These events are triggered when a device is being initialized by the OpenACC runtime. This may be
3213 when the program starts, or may be later during execution when the program reaches an **acc_init**
3214 call or an OpenACC construct. The **acc_ev_device_init_start** is triggered before device
3215 initialization starts and **acc_ev_device_init_end** after initialization is complete.

3216 The *device shutdown* event names are

acc_ev_device_shutdown_start**acc_ev_device_shutdown_end**

3217 These events are triggered when a device is shut down, most likely by a call to the OpenACC
3218 **acc_shutdown** API routine. The **acc_ev_device_shutdown_start** is triggered before
3219 the device shutdown process starts and **acc_ev_device_shutdown_end** after the device shut-
3220 down is complete.

5.1.3. Enter Data and Exit Data

3221 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**

3223 The **acc_ev_enter_data_start** and **acc_ev_enter_data_end** events are triggered at
3224 **enter data** directives, entry to data constructs, and entry to implicit data regions such as those
3225 generated by compute constructs. The **acc_ev_enter_data_start** event is triggered before
3226 any *data allocation*, *data update*, or *wait* events that are associated with that directive or region
3227 entry, and the **acc_ev_enter_data_end** is triggered after those events.

3228 The **acc_ev_exit_data_start** and **acc_ev_exit_data_end** events are triggered at **exit**
3229 **data** directives, exit from **data** constructs, and exit from implicit data regions. The
3230 **acc_ev_exit_data_start** event is triggered before any *data deallocation*, *data update*, or
3231 *wait* events associated with that directive or region exit, and the **acc_ev_exit_data_end** event
3232 is triggered after those events.

3233 When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the
3234 compiler the **implicit** field in the event structure will be set to **1**. When the construct that
3235 triggers these events was specified explicitly by the application code the **implicit** field in the
3236 event structure will be set to **0**.

5.1.4. Data Allocation

3238 The *data allocation* event names are

acc_ev_create

acc_ev_delete
acc_ev_alloc
acc_ev_free

3239 An **acc_ev_alloc** event is triggered when the OpenACC runtime allocates memory from the de-
 3240 vice memory pool, and an **acc_ev_free** event is triggered when the runtime frees that memory.
 3241 An **acc_ev_create** event is triggered when the OpenACC runtime associates device memory
 3242 with local memory, such as for a data clause (**create**, **copyin**, **copy**, **copyout**) at entry to
 3243 a data construct, compute construct, at an **enter data** directive, or in a call to a data API rou-
 3244 tine (**acc_copyin**, **acc_create**, ...). An **acc_ev_create** event may be preceded by an
 3245 **acc_ev_alloc** event, if newly allocated memory is used for this device data, or it may not, if
 3246 the runtime manages its own memory pool. An **acc_ev_delete** event is triggered when the
 3247 OpenACC runtime disassociates device memory from local memory, such as for a data clause at
 3248 exit from a data construct, compute construct, at an **exit data** directive, or in a call to a data API
 3249 routine (**acc_copyout**, **acc_delete**, ...). An **acc_ev_delete** event may be followed by
 3250 an **acc_ev_free** event, if the disassociated device memory is freed, or it may not, if the runtime
 3251 manages its own memory pool.

3252 When the action that generates a *data allocation* event was generated explicitly by the application
 3253 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event
 3254 is triggered because of a variable or array with implicitly-determined data attributes or otherwise
 3255 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

3256 5.1.5. Data Construct

3257 The events for entering and leaving *data constructs* are mapped to *enter data* and *exit data* events
 3258 as described in Section 5.1.3 Enter Data and Exit Data.

3259 5.1.6. Update Directive

3260 The *update directive* event names are

acc_ev_update_start
acc_ev_update_end

3261 The **acc_ev_update_start** event will be triggered at an **update** directive, before any *data*
 3262 *update* or *wait* events that are associated with the update directive are carried out, and the corre-
 3263 sponding **acc_ev_update_end** event will be triggered after any of the associated events.

3264 5.1.7. Compute Construct

3265 The *compute construct* event names are

acc_ev_compute_construct_start
acc_ev_compute_construct_end

3266 The **acc_ev_compute_construct_start** event is triggered at entry to a compute construct,
3267 before any *launch* events that are associated with entry to the compute construct. The
3268 **acc_ev_compute_construct_end** event is triggered at the exit of the compute construct,
3269 after any *launch* events associated with exit from the compute construct. If there are data clauses
3270 on the compute construct, those data clauses may be treated as part of the compute construct, or as
3271 part of a data construct containing the compute construct. The callbacks for data clauses must use
3272 the same line numbers as for the compute construct events.

3273 5.1.8. Enqueue Kernel Launch

3274 The *launch* event names are

```
acc_ev_enqueue_launch_start
acc_ev_enqueue_launch_end
```

3275 The **acc_ev_enqueue_launch_start** event is triggered just before an accelerator compu-
3276 tation is enqueued for execution on a device, and **acc_ev_enqueue_launch_end** is trig-
3277 gered just after the computation is enqueued. Note that these events are synchronous with the
3278 local thread enqueueing the computation to a device, not with the device executing the compu-
3279 tation. The **acc_ev_enqueue_launch_start** event callback routine is invoked just before
3280 the computation is enqueued, not just before the computation starts execution. More importantly,
3281 the **acc_ev_enqueue_launch_end** event callback routine is invoked after the computation is
3282 enqueued, not after the computation finished executing.

3283 **Note:** Measuring the time between the start and end launch callbacks is often unlikely to be useful,
3284 since it will only measure the time to manage the launch queue, not the time to execute the code on
3285 the device.

3286 5.1.9. Enqueue Data Update (Upload and Download)

3287 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
```

3288 The **_start** events are triggered just before each upload (data copy from local memory to device
3289 memory) operation is or download (data copy from device memory to local memory) operation is
3290 enqueued for execution on a device. The corresponding **_end** events are triggered just after each
3291 upload or download operation is enqueued.

3292 **Note:** Measuring the time between the start and end update callbacks is often unlikely to be useful,
3293 since it will only measure the time to manage the enqueue operation, not the time to perform the
3294 actual upload or download.

3295 When the action that generates a *data update* event was generated explicitly by the application
3296 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event

3297 is triggered because of a variable or array with implicitly-determined data attributes or otherwise
 3298 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

3299 5.1.10. Wait

3300 The *wait* event names are

```
acc_ev_wait_start
acc_ev_wait_end
```

3301 An **acc_ev_wait_start** will be triggered for each relevant queue before the local thread waits
 3302 for that queue to be empty. A **acc_ev_wait_end** will be triggered for each relevant queue after
 3303 the local thread has determined that the queue is empty.

3304 Wait events occur when the local thread and a device synchronize, either due to a **wait** directive
 3305 or by a *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit**
 3306 **data**, or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive
 3307 or *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the
 3308 **implicit** field in the event structure will be **1**.

3309 The OpenACC runtime need not trigger *wait* events for queues that have not been used in the
 3310 program, and need not trigger *wait* events for queues that have not been used by this thread since
 3311 the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on
 3312 all queues. If the program only uses the default (synchronous) queue and the queue associated with
 3313 **async(1)** and **async(2)** then an **acc wait** directive may trigger *wait* events only for those
 3314 three queues. If the implementation knows that no activities have been enqueued on the **async(2)**
 3315 queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for
 3316 the default queue and the **async(1)** queue.

3317 5.2. Callbacks Signature

3318 This section describes the signature of event callbacks. All event callbacks have the same signature.
 3319 The routine prototypes are available in the header file **acc_prof.h**, which is delivered with the
 3320 OpenACC implementation.

3321 All callback routines have three arguments. The first argument is a pointer to a struct containing
 3322 general information; the same struct type is used for all callback events. The second argument is
 3323 a pointer to a struct containing information specific to that callback event; there is one struct type
 3324 containing information for data events, another struct type containing information for kernel launch
 3325 events, and a third struct type for other events, containing essentially no information. The third
 3326 argument is a pointer to a struct containing information about the application programming interface
 3327 (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific
 3328 information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can
 3329 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*);
```


3330 In the descriptions, the datatype **ssize_t** means a signed 32-bit integer for a 32-bit binary and
 3331 a 64-bit integer for a 64-bit binary, the datatype **size_t** means an unsigned 32-bit integer for a
 3332 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype **int** means a 32-bit integer
 3333 for both 32-bit and 64-bit binaries. A null pointer is the pointer with value zero.

3334 5.2.1. First Argument: General Information

3335 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;
```

3336 The fields are described below.

3337 • **acc_event_t event_type** - The event type that triggered this callback. The datatype
 3338 is the enumeration type **acc_event_t**, described in the previous section. This allows the
 3339 same callback routine to be used for different events.

3340 • **int valid_bytes** - The number of valid bytes in this struct. This allows a library to inter-
 3341 face with newer runtimes that may add new fields to the struct at the end while retaining com-
 3342 patibility with older runtimes. A runtime must fill in the **event_type** and **valid_bytes**
 3343 fields, and must fill in values for all fields with offset less than **valid_bytes**. The value of
 3344 **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)
```

3345 • **int version** - A version number; the value of **_OPENACC**.

3346 • **acc_device_t device_type** - The device type corresponding to this event. The datatype
 3347 is **acc_device_t**, an enumeration type of all the supported device types, defined in **openacc.h**.

3348 • **int device_number** - The device number. Each device is numbered, typically starting at
 3349 device zero. For applications that use more than one device type, the device numbers may be
 3350 unique across all devices or may be unique only across all devices of the same device type.

3351 • **int thread_id** - The host thread ID making the callback. Host threads are given unique
 3352 thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP
 3353 thread number.

- 3354 • **ssize_t async** - The value of the **async()** clause for the directive that triggered this
3355 callback.
- 3356 • **ssize_t async_queue** - If the runtime uses a limited number of asynchronous queues,
3357 this field contains the internal asynchronous queue number used for the event.
- 3358 • **const char* src_file** - A pointer to null-terminated string containing the name of or
3359 path to the source file, if known, or a null pointer if not. If the library wants to save the source
3360 file name, it should allocate memory and copy the string.
- 3361 • **const char* func_name** - A pointer to a null-terminated string containing the name of
3362 the function in which the event occurred, if known, or a null pointer if not. If the library wants
3363 to save the function name, it should allocate memory and copy the string.
- 3364 • **int line_no** - The line number of the directive or program construct or the starting line
3365 number of the OpenACC construct corresponding to the event. A negative or zero value
3366 means the line number is not known.
- 3367 • **int end_line_no** - For an OpenACC construct, this contains the line number of the end
3368 of the construct. A negative or zero value means the line number is not known.
- 3369 • **int func_line_no** - The line number of the first line of the function named in **func_name**.
3370 A negative or zero value means the line number is not known.
- 3371 • **int func_end_line_no** - The last line number of the function named in **func_name**.
3372 A negative or zero value means the line number is not known.

3373 5.2.2. Second Argument: Event-Specific Information

3374 The second argument is a pointer to the **acc_event_info** union type.

```

3375     typedef union acc_event_info{
3376         acc_event_t event_type;
3377         acc_data_event_info data_event;
3378         acc_launch_event_info launch_event;
3379         acc_other_event_info other_event;
3380     }acc_event_info;

```

3375 The **event_type** field selects which union member to use. The first five members of each union
3376 member are identical. The second through fifth members of each union member (**valid_bytes**,
3377 **parent_construct**, **implicit**, and **tool_info**) have the same semantics for all event
3378 types:

- 3379 • **int valid_bytes** - The number of valid bytes in the respective struct. (This field is similar
3380 used as discussed in Section 5.2.1 First Argument: General Information.)
- 3381 • **acc_construct_t parent_construct** - This field describes the type of construct
3382 that caused the event to be emitted. The possible values for this field are defined by the
3383 **acc_construct_t** enum, described at the end of this section.
- 3384 • **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at
3385 a synchronous data construct or synchronous enter data, exit data or update directive. This

3386 field is set to zero when the event is triggered by an explicit directive or call to a runtime API
3387 routine.

3388 • **void* tool_info** - This field is used to pass tool-specific information from a **_start**
3389 event to the matching **_end** event. For a **_start** event callback, this field will be initialized
3390 to a null pointer. The value of this field for a **_end** event will be the value returned by
3391 the library in this field from the matching **_start** event callback, if there was one, or null
3392 otherwise. For events that are neither **_start** or **_end** events, this field will be null.

3393 Data Events

3394 For a data event, as noted in the event descriptions, the second argument will be a pointer to the
3395 **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;
```

3396 The fields specific for a data event are:

3397 • **acc_event_t event_type** - The event type that triggered this callback. The events that
3398 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
```

3399 • **const char* var_name** - A pointer to null-terminated string containing the name of the
3400 variable for which this event is triggered, if known, or a null pointer if not. If the library wants
3401 to save the variable name, it should allocate memory and copy the string.

3402 • **size_t bytes** - The number of bytes for the data event.

3403 • **const void* host_ptr** - If available and appropriate for this event, this is a pointer to
3404 the host data.

3405 • **const void* device_ptr** - If available and appropriate for this event, this is a pointer
3406 to the corresponding device data.

3407 **Launch Events**

3408 For a launch event, as noted in the event descriptions, the second argument will be a pointer to the
 3409 **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;

```

3410 The fields specific for a launch event are:

3411 • **acc_event_t event_type** - The event type that triggered this callback. The events that
 3412 use the **acc_launch_event_info** struct are:

```

    acc_ev_enqueue_launch_start
    acc_ev_enqueue_launch_end

```

3413 • **const char* kernel_name** - A pointer to null-terminated string containing the name of
 3414 the kernel being launched, if known, or a null pointer if not. If the library wants to save the
 3415 kernel name, it should allocate memory and copy the string.

3416 • **size_t num_gangs, num_workers, vector_length** - The number of gangs, work-
 3417 ers and vector lanes created for this kernel launch.

3418 **Other Events**

3419 For any event that does not use the **acc_data_event_info** or **acc_launch_event_info**
 3420 struct, the second argument to the callback routine will be a pointer to **acc_other_event_info**
 3421 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;

```

3422 **Parent Construct Enumeration**

3423 All event structures contain a **parent_construct** member that describes the type of construct
 3424 that caused the event to be emitted. The purpose of this field is to provide a means to identify

3425 the type of construct emitting the event in the cases where an event may be emitted by multi-
 3426 ple construct types, such as is the case with data and wait events. The possible values for the
 3427 **parent_construct** field are defined in the enumeration type **acc_construct_t**. In the
 3428 case of combined directives, the outermost construct of the combined construct should be specified
 3429 as the **parent_construct**. If the event was emitted as the result of the application making a
 3430 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 = 1,
    acc_construct_loop = 2,
    acc_construct_data = 3,
    acc_construct_enter_data = 4,
    acc_construct_exit_data = 5,
    acc_construct_host_data = 6,
    acc_construct_atomic = 7,
    acc_construct_declare = 8,
    acc_construct_init = 9,
    acc_construct_shutdown = 10,
    acc_construct_set = 11,
    acc_construct_update = 12,
    acc_construct_routine = 13,
    acc_construct_wait = 14,
    acc_construct_runtime_api = 15,
    acc_construct_serial = 16
}acc_construct_t;

```

3431 5.2.3. Third Argument: API-Specific Information

3432 The third argument is a pointer to the **acc_api_info** struct type, shown here.

```

typedef struct 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;

```

3433 The fields are described below:

- 3434 • **acc_device_api device_api** - The API in use for this device. The data type is the
 3435 enumeration **acc_device_api**, which is described later in this section.
- 3436 • **int valid_bytes** - The number of valid bytes in this struct. See the discussion above in
 3437 Section 5.2.1 First Argument: General Information.

- 3438 • **acc_device_t device_type** - The device type; the datatype is **acc_device_t**, de-
3439 fined in **openacc.h**.
- 3440 • **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to deter-
3441 mine the value used by that vendor's runtime.
- 3442 • **const void* device_handle** - If applicable, this will be a pointer to the API-specific
3443 device information.
- 3444 • **const void* context_handle** - If applicable, this will be a pointer to the API-specific
3445 context information.
- 3446 • **const void* async_handle** - If applicable, this will be a pointer to the API-specific
3447 async queue information.

3448 According to the value of **device_api** a library can cast the pointers of the fields **device_handle**,
3449 **context_handle** and **async_handle** to the respective device API type. The following device
3450 APIs are defined in the interface below. Any implementation-defined device API type must have a
3451 value greater than **acc_device_api_implementation_defined**.

```

typedef enum acc_device_api{
    acc_device_api_none = 0,                /* no device API */
    acc_device_api_cuda = 1,               /* CUDA driver API */
    acc_device_api_opencl = 2,            /* OpenCL API */
    acc_device_api_other = 4,             /* other device API */
    acc_device_api_implementation_defined = 1000 /* other device API */
}acc_device_api;

```

3452 5.3. Loading the Library

3453 This section describes how a tools library is loaded when the program is run. Four methods are
3454 described.

- 3455 • A tools library may be linked with the program, as any other library is linked, either as a
3456 static library or a dynamic library, and the runtime will call a predefined library initialization
3457 routine that will register the event callbacks.
- 3458 • The OpenACC runtime implementation may support a dynamic tools library, such as a shared
3459 object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime
3460 under control of the environment variable **ACC_PROFLIB**.
- 3461 • Some implementations where the OpenACC runtime is itself implemented as a dynamic li-
3462 brary may support adding a tools library using the **LD_PRELOAD** feature in Linux.
- 3463 • A tools library may be linked with the program, as in the first option, and the application itself
3464 can call a library initialization routine that will register the event callbacks.

3465 Callbacks are registered with the runtime by calling **acc_prof_register** for each event as
3466 described in Section 5.4 Registering Event Callbacks. The prototype for **acc_prof_register**
3467 is:

```
extern void acc_prof_register
```

```
(acc_event_t event_type, acc_prof_callback cb,
  acc_register_t info);
```

3468 The first argument to **acc_prof_register** is the event for which a callback is being registered
3469 (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*);
```

3470 The third argument is usually zero (or **acc_reg**). See Section 5.4.2 Disabling and Enabling Callbacks
3471 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;
```

3472 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);
```

3473 As shown in this example, the same routine (**prof_data**) can be registered for multiple events.
3474 The routine can use the **event_type** field in the **acc_prof_info** structure to determine for
3475 what event it was invoked.

3476 5.3.1. Library Registration

3477 The OpenACC runtime will invoke **acc_register_library**, passing the addresses of the reg-
3478 istration routines **acc_prof_register** and **acc_prof_unregister**, in case that routine
3479 comes from a dynamic library. In the third argument it passes the address of the lookup routine
3480 **acc_prof_lookup** to obtain the addresses of inquiry functions. No inquiry functions are de-
3481 fined in this profiling interface, but we preserve this argument for future support of sampling-based
3482 tools.

3483 Typically, the OpenACC runtime will include a *weak* definition of **acc_register_library**,
3484 which does nothing and which will be called when there is no tools library. In this case, the library
3485 can save the addresses of these routines and/or make registration calls to register any appropriate
3486 callbacks. The prototype for **acc_register_library** is:

```
extern void acc_register_library
  (acc_prof_reg reg, acc_prof_reg unreg,
  acc_prof_lookup_func lookup);
```

3487 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);

```

3488 The third argument passes the address to the lookup function **acc_prof_lookup** to obtain the
 3489 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);

```

3490 The argument of the lookup function is a string with the name of the inquiry function. There are no
 3491 inquiry functions defined for this interface.

3492 5.3.2. Statically-Linked Library Initialization

3493 A tools library can be compiled and linked directly into the application. If the library provides an
 3494 external routine **acc_register_library** as specified in Section 5.3.1 Library Registration, the
 3495 runtime will invoke that routine to initialize the library.

3496 The sequence of events is:

- 3497 1. The runtime invokes the **acc_register_library** routine from the library.
- 3498 2. The **acc_register_library** routine calls **acc_prof_register** for each event to
 3499 be monitored.
- 3500 3. **acc_prof_register** records the callback routines.
- 3501 4. The program runs, and your callback routines are invoked at the appropriate events.

3502 In this mode, only one tool library is supported.

3503 5.3.3. Runtime Dynamic Library Loading

3504 A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X,
 3505 DLL for Windows). In that case, you can have the OpenACC runtime load the library during initial-
 3506 ization. This allows you to enable runtime profiling without rebuilding or even relinking your ap-
 3507 plication. The dynamic library must implement a registration routine **acc_register_library**
 3508 as specified in Section 5.3.1 Library Registration.

3509 The user may set the environment variable **ACC_PROFLIB** to the path to the library will tell the
 3510 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
```

3511 When the OpenACC runtime initializes, it will read the **ACC_PROFLIB** environment variable (with
3512 **getenv**). The runtime will open the dynamic library (using **dlopen** or **LoadLibraryA**); if
3513 the library cannot be opened, the runtime may abort, or may continue execution with or with-
3514 out an error message. If the library is successfully opened, the runtime will get the address of
3515 the **acc_register_library** routine (using **dlsym** or **GetProcAddress**). If this routine
3516 is resolved in the library, it will be invoked passing in the addresses of the registration routine
3517 **acc_prof_register**, the deregistration routine **acc_prof_unregister**, and the lookup
3518 routine **acc_prof_lookup**. The registration routine in your library, **acc_register_library**,
3519 should register the callbacks by calling the **register** argument, and should save the addresses of
3520 the arguments (**register**, **unregister**, and **lookup**) for later use, if needed.

3521 The sequence of events is:

- 3522 1. Initialization of the OpenACC runtime.
- 3523 2. OpenACC runtime reads **ACC_PROFLIB**.
- 3524 3. OpenACC runtime loads the library.
- 3525 4. OpenACC runtime calls the **acc_register_library** routine in that library.
- 3526 5. Your **acc_register_library** routine calls **acc_prof_register** for each event to
3527 be monitored.
- 3528 6. **acc_prof_register** records the callback routines.
- 3529 7. The program runs, and your callback routines are invoked at the appropriate events.

3530 If supported, paths to multiple dynamic libraries may be specified in the **ACC_PROFLIB** environ-
3531 ment variable, separated by semicolons (;). The OpenACC runtime will open these libraries and in-
3532 voke the **acc_register_library** routine for each, in the order they appear in **ACC_PROFLIB**.

3533 5.3.4. Preloading with LD_PRELOAD

3534 The implementation may also support dynamic loading of a tools library using the **LD_PRELOAD**
3535 feature available in some systems. In such an implementation, you need only specify your tools
3536 library path in the **LD_PRELOAD** environment variable before executing your program. The Open-
3537 ACC runtime will invoke the **acc_register_library** routine in your tools library at initial-
3538 ization time. This requires that the OpenACC runtime include a dynamic library with a default
3539 (empty) implementation of **acc_register_library** that will be invoked in the normal case
3540 where there is no **LD_PRELOAD** setting. If an implementation only supports static linking, or if the
3541 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
```

3542 The sequence of events is:

- 3543 1. The operating system loader loads the library specified in **LD_PRELOAD**.
- 3544 2. The call to **acc_register_library** in the OpenACC runtime is resolved to the routine
3545 in the loaded tools library.
- 3546 3. OpenACC runtime calls the **acc_register_library** routine in that library.
- 3547 4. Your **acc_register_library** routine calls **acc_prof_register** for each event to
3548 be monitored.
- 3549 5. **acc_prof_register** records the callback routines.
- 3550 6. The program runs, and your callback routines are invoked at the appropriate events.

3551 In this mode, only a single tools library is supported, since only one **acc_register_library**
3552 initialization routine will get resolved by the dynamic loader.

3553 5.3.5. Application-Controlled Initialization

3554 An alternative to default initialization is to have the application itself call the library initialization
3555 routine, which then calls **acc_prof_register** for each appropriate event. The library may be
3556 statically linked to the application or your application may dynamically load the library.

3557 The sequence of events is:

- 3558 1. Your application calls the library initialization routine.
- 3559 2. The library initialization routine calls **acc_prof_register** for each event to be moni-
3560 tored.
- 3561 3. **acc_prof_register** records the callback routines.
- 3562 4. The program runs, and your callback routines are invoked at the appropriate events.

3563 In this mode, multiple tools libraries can be supported, with each library initialization routine in-
3564 voked by the application.

3565 5.4. Registering Event Callbacks

3566 This section describes how to register and unregister callbacks, temporarily disabling and enabling
3567 callbacks, the behavior of dynamic registration and unregistration, and requirements on an Open-
3568 ACC implementation to correctly support the interface.

3569 5.4.1. Event Registration and Unregistration

3570 The library must call the registration routine `acc_prof_register` to register each callback
3571 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_reg reg,
                        acc_prof_reg unreg, acc_prof_lookup_func lookup){
    reg(acc_ev_enqueue_upload_start, prof_data, 0);
    reg(acc_ev_enqueue_download_start, prof_data, 0);
    reg(acc_ev_enqueue_launch_start, prof_launch, 0);
}
```

3572 In this example the `prof_data` routine will be invoked for each data upload and download event,
3573 and the `prof_launch` routine will be invoked for each launch event. The `prof_data` routine
3574 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 :
            ...
    }
}
```

3575 Multiple Callbacks

3576 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);
```

3577 For most events, the callbacks will be invoked in the order in which they are registered. However,
3578 *end* events, named `acc_ev_..._end`, invoke callbacks in the reverse order. Essentially, each
3579 event has an ordered list of callback routines. A new callback routine is appended to the tail of the
3580 list for that event. For most events, that list is traversed from the head to the tail, but for *end* events,
3581 the list is traversed from the tail to the head.

3582 If a callback is registered, then later unregistered, then later still registered again, the second regis-
3583 tration is considered to be a new callback, and the callback routine will then be appended to the tail
3584 of the callback list for that event.

3585 **Unregistering**

3586 A matching call to **acc_prof_unregister** will remove that routine from the list of callback
3587 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

```

3588 Each entry on the callback list must also have a *ref* count. This keeps track of how many times
3589 this routine was added to this event's callback list. If a routine is registered *n* times, it must be
3590 unregistered *n* times before it is removed from the list. Note that if a routine is registered multiple
3591 times for the same event, its *ref* count will be incremented with each registration, but it will only be
3592 invoked once for each event instance.

3593 **5.4.2. Disabling and Enabling Callbacks**

3594 A callback routine may be temporarily disabled on the callback list for an event, then later re-
3595 enabled. The behavior is slightly different than unregistering and later re-registering that event.
3596 When a routine is disabled and later re-enabled, the routine's position on the callback list for that
3597 event is preserved. When a routine is unregistered and later re-registered, the routine's position on
3598 the callback list for that event will move to the tail of the list. Also, unregistering a callback must be
3599 done *n* times if the callback routine was registered *n* times. In contrast, disabling, and enabling an
3600 event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that
3601 routine for that event, even if it was enabled multiple times. Enabling a callback will immediately
3602 set the toggle and enable calls to that routine for that event, even if it was disabled multiple times.
3603 Registering a new callback initially sets the toggle.

3604 A call to **acc_prof_unregister** with a value of **acc_toggle** as the third argument will dis-
3605 able callbacks to the given routine. A call to **acc_prof_register** with a value of **acc_toggle**
3606 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

```

3607 A call to either **acc_prof_unregister** or **acc_prof_register** to disable or enable a call-
3608 back when that callback is not currently registered for that event will be ignored with no error.

3609 All callbacks for an event may be disabled (and re-enabled) by passing **NULL** to the second argument
3610 and **acc_toggle** to the third argument of **acc_prof_unregister** (and **acc_prof_register**).

3611 This sets a toggle for that event, which is distinct from the toggle for each callback for that event.
 3612 While the event is disabled, no callbacks for that event will be invoked. Callbacks for that event can
 3613 be registered, unregistered, enabled, and disabled while that event is disabled, but no callbacks will
 3614 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

```

3615 Finally, all callbacks can be disabled (and enabled) by passing the argument list (**0**, **NULL**,
 3616 **acc_toggle**) to **acc_prof_unregister** (and **acc_prof_register**). This sets a global
 3617 toggle disabling all callbacks, which is distinct from the toggle enabling callbacks for each event and
 3618 the toggle enabling each callback routine. The behavior of passing zero as the first argument and a
 3619 non-**NULL** value as the second argument to **acc_prof_unregister** or **acc_prof_register**
 3620 is not defined, and may be ignored by the runtime without error.

3621 All callbacks can be disabled (or enabled) for just the current thread by passing the argument list
 3622 (**0**, **NULL**, **acc_toggle_per_thread**) to **acc_prof_unregister** (and **acc_prof_register**).
 3623 This is the only thread-specific interface to **acc_prof_register** and **acc_prof_unregister**,
 3624 all other calls to register, unregister, enable, or disable callbacks affect all threads in the application.

3625 5.5. Advanced Topics

3626 This section describes advanced topics such as dynamic registration and changes of the execution
 3627 state for callback routines as well as the runtime and tool behavior for multiple host threads.

3628 5.5.1. Dynamic Behavior

3629 Callback routines may be registered or unregistered, enabled or disabled at any point in the execution
3630 of the program. Calls may appear in the library itself, during the processing of an event. The
3631 OpenACC runtime must allow for this case, where the callback list for an event is modified while
3632 that event is being processed.

3633 Dynamic Registration and Unregistration

3634 Calls to **acc_register** and **acc_unregister** may occur at any point in the application. A
3635 callback routine can be registered or unregistered from a callback routine, either the same routine
3636 or another routine, for a different event or the same event for which the callback was invoked. If a
3637 callback routine is registered for an event while that event is being processed, then the new callback
3638 routine will be added to the tail of the list of callback routines for this event. Some events (the
3639 **_end**) events process the callback routines in reverse order, from the tail to the head. For those
3640 events, adding a new callback routine will not cause the new routine to be invoked for this instance
3641 of the event. The other events process the callback routines in registration order, from the head to
3642 the tail. Adding a new callback routine for such a event will cause the runtime to invoke that newly
3643 registered callback routine for this instance of the event. Both the runtime and the library must
3644 implement and expect this behavior.

3645 If an existing callback routine is unregistered for an event while that event is being processed, that
3646 callback routine is removed from the list of callbacks for this event. For any event, if that callback
3647 routine had not yet been invoked for this instance of the event, it will not be invoked.

3648 Registering and unregistering a callback routine is a global operation and affects all threads, in a
3649 multithreaded application. See Section 5.4.1 Multiple Callbacks.

3650 Dynamic Enabling and Disabling

3651 Calls to **acc_register** and **acc_unregister** to enable and disable a specific callback for
3652 an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur
3653 at any point in the application. A callback routine can be enabled or disabled from a callback
3654 routine, either the same routine or another routine, for a different event or the same event for which
3655 the callback was invoked. If a callback routine is enabled for an event while that event is being
3656 processed, then the new callback routine will be immediately enabled. If it appears on the list of
3657 callback routines closer to the head (for **_end** events) or closer to the tail (for other events), that
3658 newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled
3659 or unregistered before that callback is reached.

3660 If a callback routine is disabled for an event while that event is being processed, that callback routine
3661 is immediately disabled. For any event, if that callback routine had not yet been invoked for this in-
3662 stance of the event, it will not be invoked, unless it is enabled before that callback routine is reached
3663 in the list of callbacks for this event. If all callbacks for an event are disabled while that event is
3664 being processed, or all callbacks are disabled for all events while an event is being processed, then
3665 when this callback routine returns, no more callbacks will be invoked for this instance of the event.

3666 Registering and unregistering a callback routine is a global operation and affects all threads, in a
3667 multithreaded application. See Section 5.4.1 Multiple Callbacks.

3668 5.5.2. OpenACC Events During Event Processing

3669 OpenACC events may occur during event processing. This may be because of OpenACC API rou-
3670 tine calls or OpenACC constructs being reached during event processing, or because of multiple host
3671 threads executing asynchronously. Both the OpenACC runtime and the tool library must implement
3672 the proper behavior.

3673 5.5.3. Multiple Host Threads

3674 Many programs that use OpenACC also use multiple host threads, such as programs using the
3675 OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the
3676 tools library.

3677 Runtime Support for Multiple Threads

3678 The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this
3679 tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so
3680 registering a callback from one thread will cause all threads to execute that callback. This means that
3681 managing the callback lists for each event must be protected from multiple simultaneous updates.
3682 This includes adding a callback to the tail of the callback list for an event, removing a callback from
3683 the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an
3684 event.

3685 In addition, one thread may register, unregister, enable, or disable a callback for an event while
3686 another thread is processing the callback list for that event asynchronously. The exact behavior may
3687 be dependent on the implementation, but some behaviors are expected and others are disallowed.
3688 In the following examples, there are three callbacks, A, B, and C, registered for event E in that
3689 order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is
3690 dynamically modifying the callbacks for event E while thread T2 is processing an instance of event
3691 E.

- 3692 • Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not
3693 invoke callback A for this event instance, but it must invoke callback B; if it invokes callback
3694 A, that must precede the invocation of callback B.
- 3695 • Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not
3696 invoke callback B for this event instance, but it must invoke callback A; if it invokes callback
3697 B, that must follow the invocation of callback A.
- 3698 • Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback
3699 B for event E. Thread T2 may or may not invoke callback A and may or may not invoke
3700 callback B for this event instance, but if it invokes both callbacks, it must invoke callback A
3701 before it invokes callback B.
- 3702 • Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback
3703 A for event E. Thread T2 may or may not invoke callback A and may or may not invoke
3704 callback B for this event instance, but if it invokes callback B, it must have invoked callback
3705 A for this event instance.
- 3706 • Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not

3707 invoke callback D for this event instance, but it must invoke both callbacks A and B. If it
3708 invokes callback D, that must follow the invocations of A and B.

- 3709 • Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke
3710 callback C for this event instance, but it must invoke both callbacks A and B. If it invokes
3711 callback C, that must follow the invocations of A and B.

3712 The `acc_prof_info` struct has a `thread_id` field, which the runtime must set to a unique
3713 value for each host thread, though it need not be the same as the OpenMP threadnum value.

3714 **Library Support for Multiple Threads**

3715 The tool library must also be thread-safe. The callback routine will be invoked in the context of the
3716 thread that reaches the event. The library may receive a callback from a thread T2 while it's still
3717 processing a callback, from the same event type or from a different event type, from another thread
3718 T1. The `acc_prof_info` struct has a `thread_id` field, which the runtime must set to a unique
3719 value for each host thread.

3720 If the tool library uses dynamic callback registration and unregistration, or callback disabling and
3721 enabling, recall that unregistering or disabling an event callback from one thread will unregister or
3722 disable that callback for all threads, and registering or enabling an event callback from any thread
3723 will register or enable it for all threads. If two or more threads register the same callback for the
3724 same event, the behavior is the same as if one thread registered that callback multiple times; see
3725 Section 5.4.1 Multiple Callbacks. The `acc_unregister` routine must be called as many times
3726 as `acc_register` for that callback/event pair in order to totally unregister it. If two threads
3727 register two different callback routines for the same event, unless the order of the registration calls
3728 is guaranteed by some synchronization method, the order in which the runtime sees the registration
3729 may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.

6. Glossary

- 3731 Clear and consistent terminology is important in describing any programming model. We define
3732 here the terms you must understand in order to make effective use of this document and the asso-
3733 ciated programming model. In particular, some terms used in this specification conflict with their
3734 usage in the base language specifications. When there is potential confusion, the term will appear
3735 here.
- 3736 **Accelerator** – a device attached to a CPU and to which the CPU can offload data and compute
3737 kernels to perform compute-intensive calculations.
- 3738 **Accelerator routine** – a C or C++ function or Fortran subprogram compiled for the accelerator
3739 with the **routine** directive.
- 3740 **Accelerator thread** – a thread of execution that executes on the accelerator; a single vector lane of
3741 a single worker of a single gang.
- 3742 **Aggregate datatype** – an array or composite datatype, or any non-scalar datatype. In Fortran, ag-
3743 gregate datatypes include arrays and derived types. In C, aggregate datatypes include arrays, targets
3744 of pointers, structs, and unions. In C++, aggregate datatypes include arrays, targets of pointers,
3745 classes, structs, and unions.
- 3746 **Aggregate variables** – an array or composite variable, or a variable of any non-scalar datatype.
- 3747 **Async-argument** – an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++,
3748 *integer* for Fortran), or one of the special values **acc_async_noval** or **acc_async_sync**.
- 3749 **Barrier** – a type of synchronization where all parallel execution units or threads must reach the
3750 barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after
3751 the starting barrier on a horse race track.
- 3752 **Compute intensity** – for a given loop, region, or program unit, the ratio of the number of arithmetic
3753 operations performed on computed data divided by the number of memory transfers required to
3754 move that data between two levels of a memory hierarchy.
- 3755 **Construct** – a directive and the associated statement, loop, or structured block, if any.
- 3756 **Composite datatype** – a derived type in Fortran, or a **struct** or **union** type in C, or a **class**,
3757 **struct**, or **union** type in C++. (This is different from the use of the term *composite data type* in
3758 the C and C++ languages.)
- 3759 **Composite variable** – a variable of composite datatype. In Fortran, a composite variable must not
3760 have allocatable or pointer attributes.
- 3761 **Compute construct** – a *parallel construct*, *kernels construct*, or *serial construct*.
- 3762 **Compute region** – a *parallel region*, *kernels region*, or *serial region*.
- 3763 **CUDA** – the CUDA environment from NVIDIA is a C-like programming environment used to
3764 explicitly control and program an NVIDIA GPU.

- 3765 **Current device** – the device represented by the *acc-current-device-type-var* and *acc-current-device-*
3766 *num-var* ICVs
- 3767 **Current device type** – the device type represented by the *acc-current-device-type-var* ICV
- 3768 **Data lifetime** – the lifetime of a data object in device memory, which may begin at the entry to
3769 a data region, or at an **enter data** directive, or at a data API call such as **acc_copyin** or
3770 **acc_create**, and which may end at the exit from a data region, or at an **exit data** directive,
3771 or at a data API call such as **acc_delete**, **acc_copyout**, or **acc_shutdown**, or at the end of
3772 the program execution.
- 3773 **Data region** – a *region* defined by a **data** construct, or an implicit data region for a function or
3774 subroutine containing OpenACC directives. Data constructs typically allocate device memory and
3775 copy data from host to device memory upon entry, and copy data from device to local memory and
3776 deallocate device memory upon exit. Data regions may contain other data regions and compute
3777 regions.
- 3778 **Device** – a general reference to an accelerator or a multicore CPU.
- 3779 **Default asynchronous queue** – the asynchronous activity queue represented in the *acc-default-*
3780 *async-var* ICV
- 3781 **Device memory** – memory attached to a device, logically and physically separate from the host
3782 memory.
- 3783 **Device thread** – a thread of execution that executes on any device.
- 3784 **Directive** – in C or C++, a **#pragma**, or in Fortran, a specially formatted comment statement, that
3785 is interpreted by a compiler to augment information about or specify the behavior of the program.
- 3786 **Discrete memory** – memory accessible from the local thread that is not accessible from the current
3787 device, or memory accessible from the current device that is not accessible from the local thread.
- 3788 **DMA** – Direct Memory Access, a method to move data between physically separate memories;
3789 this is typically performed by a DMA engine, separate from the host CPU, that can access the host
3790 physical memory as well as an IO device or other physical memory.
- 3791 **GPU** – a Graphics Processing Unit; one type of accelerator.
- 3792 **GPGPU** – General Purpose computation on Graphics Processing Units.
- 3793 **Host** – the main CPU that in this context may have one or more attached accelerators. The host
3794 CPU controls the program regions and data loaded into and executed on one or more devices.
- 3795 **Host thread** – a thread of execution that executes on the host.
- 3796 **Implicit data region** – the data region that is implicitly defined for a Fortran subprogram or C
3797 function. A call to a subprogram or function enters the implicit data region, and a return from the
3798 subprogram or function exits the implicit data region.
- 3799 **Kernel** – a nested loop executed in parallel by the accelerator. Typically the loops are divided into
3800 a parallel domain, and the body of the loop becomes the body of the kernel.
- 3801 **Kernels region** – a *region* defined by a **kernels** construct. A kernels region is a structured block
3802 which is compiled for the accelerator. The code in the kernels region will be divided by the compiler
3803 into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region
3804 may require space in device memory to be allocated and data to be copied from local memory to

3805 device memory upon region entry, and data to be copied from device memory to local memory and
3806 space in device memory to be deallocated upon exit.

3807 **Level of parallelism** – The possible levels of parallelism in OpenACC are gang, worker, vector,
3808 and sequential. One or more of gang, worker, and vector parallelism may appear on a loop con-
3809 struct. Sequential execution corresponds to no parallelism. The **gang**, **worker**, **vector**, and
3810 **seq** clauses specify the level of parallelism for a loop.

3811 **Local device** – the device where the *local thread* executes.

3812 **Local memory** – the memory associated with the *local thread*.

3813 **Local thread** – the host thread or the accelerator thread that executes an OpenACC directive or
3814 construct.

3815 **Loop trip count** – the number of times a particular loop executes.

3816 **MIMD** – a method of parallel execution (Multiple Instruction, Multiple Data) where different exe-
3817 cution units or threads execute different instruction streams asynchronously with each other.

3818 **OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming
3819 environment that enables low-level general-purpose programming on GPUs and other accelerators.

3820 **Orphaned loop construct** - a **loop** construct that is not lexically contained in any compute con-
3821 struct, that is, that has no parent compute construct.

3822 **Parallel region** – a *region* defined by a **parallel** construct. A parallel region is a structured block
3823 which is compiled for the accelerator. A parallel region typically contains one or more work-sharing
3824 loops. A parallel region may require space in device memory to be allocated and data to be copied
3825 from local memory to device memory upon region entry, and data to be copied from device memory
3826 to local memory and space in device memory to be deallocated upon exit.

3827 **Parent compute construct** – for a **loop** construct, the **parallel**, **kernels**, or **serial** con-
3828 struct that lexically contains the **loop** construct and is the innermost compute construct that con-
3829 tains that **loop** construct, if any.

3830 **Present data** – data for which the sum of the structured and dynamic reference counters is greater
3831 than zero.

3832 **Private data** – with respect to an iterative loop, data which is used only during a particular loop
3833 iteration. With respect to a more general region of code, data which is used within the region but is
3834 not initialized prior to the region and is re-initialized prior to any use after the region.

3835 **Procedure** – in C or C++, a function in the program; in Fortran, a subroutine or function.

3836 **Region** – all the code encountered during an instance of execution of a construct. A region includes
3837 any code in called routines, and may be thought of as the dynamic extent of a construct. This may
3838 be a *parallel region*, *kernels region*, *serial region*, *data region* or *implicit data region*.

3839 **Scalar** – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer
3840 attributes.

3841 **Scalar datatype** – an intrinsic or built-in datatype that is not an array or aggregate datatype. In For-
3842 tran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes
3843 are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long at-
3844 tribute), enum, float, double, long double, `_Complex` (with optional float or long attribute), or any
3845 pointer datatype. In C++, scalar datatypes are char (signed or unsigned), `wchar_t`, int (signed or

- 3846 unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double,
3847 or any pointer datatype. Not all implementations or targets will support all of these datatypes.
- 3848 **Serial region** – a *region* defined by a **serial** construct. A serial region is a structured block which
3849 is compiled for the accelerator. A serial region contains code that is executed by one vector lane of
3850 one worker in one gang. A serial region may require space in device memory to be allocated and
3851 data to be copied from local memory to device memory upon region entry, and data to be copied
3852 from device memory to local memory and space in device memory to be deallocated upon exit.
- 3853 **Shared memory** – memory that is accessible from both the local thread and the current device.
- 3854 **SIMD** – A method of parallel execution (single-instruction, multiple-data) where the same instruc-
3855 tion is applied to multiple data elements simultaneously.
- 3856 **SIMD operation** – a *vector operation* implemented with SIMD instructions.
- 3857 **Structured block** – in C or C++, an executable statement, possibly compound, with a single entry
3858 at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single
3859 entry at the top and a single exit at the bottom.
- 3860 **Thread** – On a host CPU, a thread is defined by a program counter and stack location; several host
3861 threads may comprise a process and share host memory. On an accelerator, a thread is any one
3862 vector lane of one worker of one gang.
- 3863 **var** – the name of a variable (scalar, array, or composite variable), or a subarray specification, or an
3864 array element, or a composite variable member, or the name of a Fortran common block between
3865 slashes.
- 3866 **Vector operation** – a single operation or sequence of operations applied uniformly to each element
3867 of an array.
- 3868 **Visible device copy** – a copy of a variable, array, or subarray allocated in device memory that is
3869 visible to the program unit being compiled.

3870 **A. Recommendations for Implementors**

3871 This section gives recommendations for standard names and extensions to use for implementations
3872 for specific targets and target platforms, to promote portability across such implementations, and
3873 recommended options that programmers find useful. While this appendix is not part of the Open-
3874 ACC specification, implementations that provide the functionality specified herein are strongly rec-
3875 ommended to use the names in this section. The first subsection describes devices, such as NVIDIA
3876 GPUs. The second subsection describes additional API routines for target platforms, such as CUDA
3877 and OpenCL. The third subsection lists several recommended options for implementations.

3878 **A.1. Target Devices**

3879 **A.1.1. NVIDIA GPU Targets**

3880 This section gives recommendations for implementations that target NVIDIA GPU devices.

3881 **Accelerator Device Type**

3882 These implementations should use the name **acc_device_nvidia** for the **acc_device_t**
3883 type or return values from OpenACC Runtime API routines.

3884 **ACC_DEVICE_TYPE**

3885 An implementation should use the case-insensitive name **nvidia** for the environment variable
3886 **ACC_DEVICE_TYPE**.

3887 **device.type clause argument**

3888 An implementation should use the case-insensitive name **nvidia** as the argument to the **device.type**
3889 clause.

3890 **A.1.2. AMD GPU Targets**

3891 This section gives recommendations for implementations that target AMD GPUs.

3892 **Accelerator Device Type**

3893 These implementations should use the name **acc_device_radeon** for the **acc_device_t**
3894 type or return values from OpenACC Runtime API routines.

3895 **ACC_DEVICE_TYPE**

3896 These implementations should use the case-insensitive name **radeon** for the environment variable
3897 **ACC_DEVICE_TYPE**.

3898 **device_type clause argument**

3899 An implementation should use the case-insensitive name **radeon** as the argument to the **device_type**
3900 clause.

3901 **A.1.3. Multicore Host CPU Target**

3902 This section gives recommendations for implementations that target the multicore host CPU.

3903 **Accelerator Device Type**

3904 These implementations should use the name **acc_device_host** for the **acc_device_t** type
3905 or return values from OpenACC Runtime API routines.

3906 **ACC_DEVICE_TYPE**

3907 These implementations should use the case-insensitive name **host** for the environment variable
3908 **ACC_DEVICE_TYPE**.

3909 **device_type clause argument**

3910 An implementation should use the case-insensitive name **host** as the argument to the **device_type**
3911 clause.

3912 **A.2. API Routines for Target Platforms**

3913 These runtime routines allow access to the interface between the OpenACC runtime API and the
3914 underlying target platform. An implementation may not implement all these routines, but if it
3915 provides this functionality, it should use these function names.

3916 A.2.1. NVIDIA CUDA Platform

3917 This section gives runtime API routines for implementations that target the NVIDIA CUDA Run-
3918 time or Driver API.

3919 **acc_get_current_cuda_device**

3920 **Summary** The **acc_get_current_cuda_device** routine returns the NVIDIA CUDA de-
3921 vice handle for the current device.

3922 **Format**

C or C++:

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

3923 **acc_get_current_cuda_context**

3924 **Summary** The **acc_get_current_cuda_context** routine returns the NVIDIA CUDA
3925 context handle in use for the current device.

3926 **Format**

C or C++:

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

3927 **acc_get_cuda_stream**

3928 **Summary** The **acc_get_cuda_stream** routine returns the NVIDIA CUDA stream handle in
3929 use for the current device for the asynchronous activity queue associated with the **async** argument.
3930 This argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.

3931 **Format**

C or C++:

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

3932 **acc_set_cuda_stream**

3933 **Summary** The **acc_set_cuda_stream** routine sets the NVIDIA CUDA stream handle the
3934 current device for the asynchronous activity queue associated with the **async** argument. This
3935 argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.

3936 **Format**

C or C++:

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

3937 **A.2.2. OpenCL Target Platform**

3938 This section gives runtime API routines for implementations that target the OpenCL API on any
3939 device.

3940 **acc_get_current_opengl_device**

3941 **Summary** The `acc_get_current_opengl_device` routine returns the OpenCL device
3942 handle for the current device.

3943 **Format**

C or C++:

```
void* acc_get_current_opengl_device ();
```

3944 **acc_get_current_opengl_context**

3945 **Summary** The `acc_get_current_opengl_context` routine returns the OpenCL context
3946 handle in use for the current device.

3947 **Format**

C or C++:

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

3948 **acc_get_opengl_queue**

3949 **Summary** The `acc_get_opengl_queue` routine returns the OpenCL command queue han-
3950 dle in use for the current device for the asynchronous activity queue associated with the **async**
3951 argument. This argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.

3952 **Format**

C or C++:

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


3953 **acc_set_opencl_queue**

3954 **Summary** The `acc_set_opencl_queue` routine returns the OpenCL command queue handle in use for the current device for the asynchronous activity queue associated with the `async` argument. This argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.

3957 **Format**

C or C++:

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

3958 **A.3. Recommended Options**

3959 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.

3961 **A.3.1. C Pointer in Present clause**

3962 This revision of OpenACC clarifies the construct:

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

3963 This example tests whether the pointer `p` itself is present in the current device memory. Implementations before this revision commonly implemented this by testing whether the pointer target `p[0]` was present in the current device memory, and this appears in many programs assuming such. Until such programs are modified to comply with this revision, an option to implement `present(p)` as `present(p[0])` for C pointers may be helpful to users.

3968 **A.3.2. Autoscopying**

3969 If an implementation implements autoscopying to automatically determine variables that are private to a compute region or to a loop, or to recognize reductions in a compute region or a loop, an option to print a message telling what variables were affected by the analysis would be helpful to users. An option to disable the autoscopying analysis would be helpful to promote program portability across implementations.

Index

- 3974 **_OPENACC**, 16–20, 24, 121
- 3975 acc-current-device-num-var, 24
- 3976 acc-current-device-type-var, 24
- 3977 acc-default-async-var, 24, 81
- 3978 **acc_async_noval**, 16, 81
- 3979 **acc_async_sync**, 16, 81
- 3980 **acc_device_host**, 142
- 3981 **ACC_DEVICE_NUM**, 25, 113
- 3982 **acc_device_nvidia**, 141
- 3983 **acc_device_radeon**, 142
- 3984 **ACC_DEVICE_TYPE**, 25, 113, 141, 142
- 3985 **ACC_PROFLIB**, 113
- 3986 action
 - 3987 attach, 41, 45
 - 3988 copyin, 44
 - 3989 copyout, 44
 - 3990 create, 44
 - 3991 delete, 45
 - 3992 detach, 41, 45
 - 3993 immediate, 46
 - 3994 present decrement, 44
 - 3995 present increment, 43
- 3996 AMD GPU target, 141
- 3997 **async** clause, 40, 76, 80
- 3998 async queue, 11
- 3999 *async-argument*, 81
- 4000 asynchronous execution, 11, 80
- 4001 **atomic** construct, 16, 63
- 4002 attach action, 41, 45
- 4003 **attach** clause, 50
- 4004 attachment counter, 41
- 4005 **auto** clause, 16, 55
- 4006 autoscoping, 145
- 4007 barrier synchronization, 11, 28, 29, 31, 137
- 4008 **bind** clause, 79
- 4009 **cache** directive, 61
- 4010 **capture** clause, 67
- 4011 **collapse** clause, 53
- 4012 common block, 41, 68, 70, 80
- 4013 compute construct, 137
- 4014 compute region, 137
- 4015 construct, 137
 - 4016 **atomic**, 63
 - 4017 compute, 137
 - 4018 **data**, 37, 41
 - 4019 **host_data**, 51
 - 4020 **kernels**, 28, 41
 - 4021 **kernels loop**, 62
 - 4022 **parallel**, 27, 41
 - 4023 **parallel loop**, 62
 - 4024 **serial**, 30, 41
 - 4025 **serial loop**, 62
- 4026 **copy** clause, 47
- 4027 copyin action, 44
- 4028 **copyin** clause, 47
- 4029 copyout action, 44
- 4030 **copyout** clause, 48
- 4031 create action, 44
- 4032 **create** clause, 49, 69
- 4033 CUDA, 11, 12, 137, 141, 143
- 4034 data attribute
 - 4035 explicitly determined, 35
 - 4036 implicitly determined, 35
 - 4037 predetermined, 35
- 4038 data clause, 41
- 4039 **data** construct, 37, 41
- 4040 data lifetime, 138
- 4041 data region, 36, 138
 - 4042 implicit, 36
- 4043 **declare** directive, 16, 67
- 4044 **default** clause, 34
- 4045 **default (none)** clause, 16, 28, 29, 31
- 4046 default(present), 28, 29, 31
- 4047 delete action, 45
- 4048 **delete** clause, 50
- 4049 detach action, 41, 45
 - 4050 immediate, 46

- 4051 **detach** clause, 51
- 4052 **device** clause, 75
- 4053 **device_resident** clause, 69
- 4054 **device_type** clause, 25
- 4055 **device_type** clause, 16, 41, 141, 142
- 4056 **deviceptr** clause, 41, 46
- 4057 direct memory access, 11, 138
- 4058 DMA, 11, 138
- 4059 **enter data** directive, 38, 41
- 4060 environment variable
- 4061 **_OPENACC**, 24
- 4062 **ACC_DEVICE_NUM**, 25, 113
- 4063 **ACC_DEVICE_TYPE**, 25, 113, 141, 142
- 4064 **ACC_PROFLIB**, 113
- 4065 **exit data** directive, 38, 41
- 4066 explicitly determined data attribute, 35
- 4067 **firstprivate** clause, 28, 31, 33
- 4068 gang, 27, 31
- 4069 **gang** clause, 54, 78
- 4070 gang parallelism, 10
- 4071 *gang-arg*, 53
- 4072 gang-partitioned mode, 10
- 4073 gang-redundant mode, 10, 27, 31
- 4074 GP mode, 10
- 4075 GR mode, 10
- 4076 **host**, 142
- 4077 **host** clause, 16, 75
- 4078 **host_data** construct, 51
- 4079 ICV, 24
- 4080 **if** clause, 38, 39, 71, 73, 74, 76, 83
- 4081 immediate detach action, 46
- 4082 implicit data region, 36
- 4083 implicitly determined data attribute, 35
- 4084 **independent** clause, 56
- 4085 **init** directive, 71
- 4086 internal control variable, 24
- 4087 **kernels** construct, 28, 41
- 4088 **kernels loop** construct, 62
- 4089 level of parallelism, 10, 139
- 4090 **link** clause, 16, 41, 70
- 4091 local device, 11
- 4092 local memory, 11
- 4093 local thread, 11
- 4094 **loop** construct, 52
- 4095 orphaned, 53
- 4096 **no_create** clause, 49
- 4097 **nohost** clause, 80
- 4098 **num_gangs** clause, 32
- 4099 **num_workers** clause, 32
- 4100 **nvidia**, 141
- 4101 NVIDIA GPU target, 141
- 4102 OpenCL, 11, 12, 139, 141, 144
- 4103 orphaned **loop** construct, 53
- 4104 **parallel** construct, 27, 41
- 4105 **parallel loop** construct, 62
- 4106 parallelism
- 4107 level, 10, 139
- 4108 parent compute construct, 53
- 4109 predetermined data attribute, 35
- 4110 **present** clause, 41, 46
- 4111 present decrement action, 44
- 4112 present increment action, 43
- 4113 **private** clause, 33, 57
- 4114 **radeon**, 142
- 4115 **read** clause, 67
- 4116 **reduction** clause, 33, 57
- 4117 reference counter, 40
- 4118 region
- 4119 compute, 137
- 4120 data, 36, 138
- 4121 implicit data, 36
- 4122 **routine** directive, 16, 77
- 4123 **self** clause, 16, 75
- 4124 sentinel, 23
- 4125 **seq** clause, 55, 79
- 4126 **serial** construct, 30, 41
- 4127 **serial loop** construct, 62
- 4128 **shutdown** directive, 72
- 4129 *size-expr*, 53
- 4130 thread, 140
- 4131 **tile** clause, 16, 56
- 4132 **update** clause, 67
- 4133 **update** directive, 74
- 4134 **use_device** clause, 52
- 4135 **vector** clause, 55, 79

- 4136 vector lane, 27
- 4137 vector parallelism, 10
- 4138 vector-partitioned mode, 10
- 4139 vector-single mode, 10
- 4140 **vector_length** clause, 33
- 4141 visible device copy, 140
- 4142 VP mode, 10
- 4143 VS mode, 10

- 4144 **wait** clause, 40, 76, 81
- 4145 **wait** directive, 82
- 4146 worker, 27, 31
- 4147 **worker** clause, 54, 78
- 4148 worker parallelism, 10
- 4149 worker-partitioned mode, 10
- 4150 worker-single mode, 10
- 4151 WP mode, 10
- 4152 WS mode, 10