



OpenACC

API 2.5

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator device, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C and C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

General Syntax

C/C++

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

FORTRAN

!\$acc *directive* [*clause* [,] *clause*]...

An OpenACC construct is an OpenACC directive and, if applicable, the immediately following statement, loop or structured block.

Parallel Construct

A **parallel** construct launches a number of gangs executing in parallel, where each gang may support multiple workers, each with vector or SIMD operations.

C/C++

```
#pragma acc parallel [clause [,] clause...] new-line  
{ structured block }
```

FORTRAN

```
!$acc parallel [clause [,] clause...]  
structured block  
!$acc end parallel
```

Compute Construct and Data clauses are also allowed; data clauses on the parallel construct modify the structured reference counts for the associated data.

OTHER CLAUSES

reduction(operator: list)

A private copy of each variable in *list* is allocated for each gang. The values for all gangs are combined with the *operator* at the end of the parallel region. Valid C and C++ operators are **+**, *****, **max**, **min**, **&**, **|**, **^**, **&&**, **||**. Valid Fortran operators are **+**, *****, **max**, **min**, **iand**, **ior**, **ieor**, **.and.**, **.or.**, **.eqv.**, **.neqv.**

private(list)

A copy of each variable in *list* is allocated for each gang.

firstprivate(list)

A copy of each variable in *list* is allocated for each gang and initialized with the value of the variable of the encountering thread.

Kernels Construct

A **kernels** construct surrounds loops to be executed on the device, typically as a sequence of kernel operations.

C/C++

```
#pragma acc kernels [clause [,] clause...] new-line  
{ structured block }
```

FORTRAN

```
!$acc kernels [clause [,] clause...]  
structured block  
!$acc end kernels
```

Compute Construct and Data clauses are also allowed; data clauses on the kernels construct modify the structured reference counts for the associated data.

Compute Construct Clauses

if(*condition*)

When the *condition* is nonzero or .TRUE. the kernels region will execute on the device; otherwise, the encountering thread will execute the region.

default(*none*)

Prevents the compiler from implicitly determining data attributes for any variable used or assigned in the construct.

default(*present*)

Implicitly assume any non-scalar data not specified in a data clause is present.

device_type or **dtype([* | *device-type-list*])**

May be followed by any of the clauses below. Clauses following **device_type** will apply only when compiling for the given device type(s). Clauses following **device_type(*)** apply to all devices not named in another **device_type** clause.

async [(*expression*)]

The kernels region executes asynchronously with the encountering thread on the corresponding async queue.

wait [(*expression-list*)]

The kernels region will not begin execution until all actions on the corresponding async queue(s) are complete.

num_gangs(*expression*)

Controls how many parallel gangs are created.

num_workers(*expression*)

Controls how many workers are created in each gang.

vector_length(*expression*)

Controls the vector length on each worker.

Data Construct

An device **data** construct defines a region of the program within which data is accessible by the device.

C/C++

```
#pragma acc data [clause[,] clause]... new-line  
{ structured block }
```

FORTRAN

```
!$acc data [clause[,] clause]...  
structured block  
!$acc end data
```

Data clauses are also allowed; data clauses on the data construct modify the structured reference counts for the associated data.

OTHER CLAUSES

if(*condition*)

When the *condition* is zero or .FALSE. no data will be allocated or moved to or from the device.

Enter Data Directive

An **enter data** directive is used to allocate and move data to the device memory for the remainder of the program, or until a matching **exit data** directive deallocates the data.

C/C++

```
#pragma acc enter data [clause[,] clause...] new-line
```

FORTRAN

```
!$acc enter data [clause[,] clause]...
```

CLAUSES

if(condition)

When the *condition* is zero or .FALSE. no data will be allocated or moved to the device.

async [(expression)]

The data movement executes asynchronously with the encountering thread on the corresponding async queue.

wait [(expression-list)]

The data movement will not begin execution until all actions on the corresponding async queue(s) are complete.

copyin(list)

create(list)

See Data Clauses; data clauses on the enter data directive modify the dynamic reference counts for the associated data.

Exit Data Directive

For data that was created with the **enter data** directive, the **exit data** directive moves data from device memory and deallocates the memory,

C/C++

```
#pragma acc exit data [clause[,] clause...] new-line
```

FORTRAN

```
!$acc exit data [clause[,] clause]...
```

CLAUSES

if(condition)

When the *condition* is zero or .FALSE. no data will be moved from the device or deallocated.

async [(expression)]

The data movement executes asynchronously with the encountering thread on the corresponding async queue.

wait [(expression-list)]

The data movement will not begin execution until all actions on the corresponding async queue(s) are complete.

finalize

Sets the dynamic reference count to zero.

copyout(list)

delete(list)

See Data Clauses; data clauses on the `exit data` directive modify the dynamic reference counts for the associated data.

Data Clauses

The description applies to the clauses used on parallel constructs, kernels constructs, data constructs, and **enter data** and **exit data** directives. Data clauses may not follow a **device_type** clause. The **copy**, **copyin**, **copyout**, **create**, **delete** and **present** clauses have no effect on a shared memory device.

copy(list) parallel, kernels, data, declare

When entering the region, if the data in *list* is already present on the current device, the structured reference count is incremented and that copy is used. Otherwise, it allocates device memory and copies the values from the encountering thread and sets the structured reference count to one. When exiting the region, the structured reference count is decremented. If both reference counts are zero, the data is copied from device memory to the encountering thread and the device memory is deallocated.

copyin(list) parallel, kernels, data, declare

When entering the region or at an **enter data** directive, if the data in *list* is already present on the current device, the appropriate reference count is incremented and that copy is used. Otherwise, it allocates device memory and copies the values from the encountering thread and sets the appropriate reference count to one. When exiting the region the structured reference count is decremented. If both reference counts are zero, the device memory is deallocated.

copyout(list) parallel, kernels, exit data, declare

When entering the region, if the data in *list* is already present on the current device, the structured reference count is incremented and that copy is used. Otherwise, it allocates device memory and sets the structured reference count to one. At an **exit data** directive with no **finalize** clause or when exiting the region, the appropriate reference count is decremented. At an **exit data** directive with a **finalize** clause, the dynamic reference count is set to zero. In any case, if both reference counts are zero, the data is copied from device memory to the encountering thread and the device memory is deallocated.

create(list) parallel, kernels, enter data, declare

When entering the region or at an **enter data** directive, if the data in *list* is already present on the current device, the appropriate reference count is incremented and that copy is used. Otherwise, it allocates device memory and sets the appropriate reference count to one. When exiting the region, the structured reference count is decremented. If both reference counts are zero, the device memory is deallocated.

delete(list) exit data

With no **finalize** clause, the dynamic reference count is decremented. With a **finalize** clause, the dynamic reference count

is set to zero. In either case, if both reference counts are zero, the device memory is deallocated.

present(*list*) parallel, kernels

When entering the region, the data must be present in device memory, and the structured reference count is incremented.

When exiting the region, the structured reference count is decremented.

deviceptr(*list*) parallel, kernels, data, declare

C and C++; the *list* entries must be pointer variables that contain device addresses, such as from **acc_malloc**. Fortran: the *list* entries must be dummy arguments, and must not have the pointer, allocatable or value attributes.

Host Data Construct

A **host_data** construct makes the address of device data available on the host.

C/C++

```
#pragma acc host_data [clause[[,] clause...] new-line  
{ structured block }
```

FORTRAN

```
!$acc host_data [clause[[,] clause...] new-line  
structured block  
!$acc end host_data
```

CLAUSES

use_device(*list*)

Directs the compiler to use the device address of any entry in *list*, for instance, when passing a variable to a procedure.

Loop Construct

A **loop** construct applies to the immediately following loop or tightly nested loops, and describes the type of device parallelism to use to execute the iterations of the loop.

C/C++

```
#pragma acc loop [clause [[,] clause]...] new-line
```

FORTRAN

```
!$acc loop [clause [[,] clause]...] new-line
```

CLAUSES

collapse(*n*)

Applies the associated directive to the following *n* tightly nested loops.

seq

Executes the loop or loops sequentially.

auto

Instructs the compiler to analyze the loop or loops to determine whether it can be safely executed in parallel, and if so, to apply gang, worker or vector parallelism.

independent

Specifies that the loop iterations are data-independent and can be executed in parallel, overriding compiler dependence analysis.

tile(*expression-list*)

With *n* expressions, specifies that the following *n* tightly nested loops should be split into *n* outer tile loops and *n* inner element loops, where the trip counts of the element loops are taken from the expression.

The first entry applies to the innermost element loop.

May be combined with one or two of **gang**, **worker** and **vector** clauses.

device_type or **dtype([***|*device-type-list*])**

May be followed by the **gang**, **worker**, **vector**, **seq**, **auto**, **tile**, and **collapse** clauses. Clauses following **device_type** will apply only when compiling for the given device type(s).

private(*list*)

A copy of each variable in *list* is created for each thread that executes the loop or loops.

reduction(*operator: list*)

A private copy of each variable in *list* is allocated for each thread that executes the loop or loops. The values for all threads are combined with the *operator* at the end of the loops. See **reduction** clause for the parallel construct for valid operators.

LOOP CLAUSES WITHIN A PARALLEL CONSTRUCT OR ORPHANED LOOP DIRECTIVE

gang

Shares the iterations of the loop or loops across the gangs of the parallel region.

worker

Shares the iterations of the loop or loops across the workers of the gang.

vector

Executes the iterations of the loop or loops in SIMD or vector mode.

LOOP CLAUSES WITHIN KERNELS CONSTRUCT

gang [(*num_gangs*)]

Executes the iterations of the loop or loops in parallel across at most *num_gangs* gangs.

worker [(*num_workers*)]

Executes the iterations of the loop or loops in parallel across at most *num_workers* workers of a single gang.

vector [(*vector_length*)]

Executes the iterations of the loop or loops in SIMD or vector mode, with a maximum *vector_length*.

Cache Directive

A **cache** directive may be added at the top of a partitioned loop. The elements or subarrays in the *list* are cached in the software-managed data cache.

C/C++

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

FORTRAN

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

Atomic Directive

The atomic construct ensures that a specific storage location is accessed or updated atomically, preventing simultaneous, conflicting reading and writing threads.

C/C++

```
#pragma acc atomic [ read | write | update |  
capture ] new-line
```

atomic-block

If no clause is specified, the **update** clause is assumed.

The atomic-block must be one of the following:

<u>clause</u>	<u>atomic-block</u>
read	$v = x;$
write	$x = \text{expr};$
update	$\text{update-expr};$
capture	$v = \text{update-expr};$ $\{ \text{update-expr}; v = x; \}$ $\{ v = x; \text{update-expr}; \}$ $\{ v = x; x = \text{expr}; \}$

where *update-expr* is one of

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

FORTRAN

```
!$acc atomic [ read | write | update | capture ]  
stmt-1  
[stmt-2]  
[ !$acc end atomic ]
```

If no clause is specified, the **update** clause is assumed. The **end atomic** directive is required if *stmt-2* is present. The statements allowed are:

<u>clause</u>	<u>stmt-1</u>	<u>stmt-2</u>
read	capture-stmt	
write	write-stmt	
update	update-stmt	
capture	update-stmt capture-stmt capture-stmt	capture-stmt update-stmt write-stmt

capture-stmt is

`v = x`

write-stmt is

`x = expr`

update-stmt is one of

`x = x operator expr`

`x = expr operator x`

`x = intrinsic_proc(x, expr-list)`

`x = intrinsic_proc(expr-list, x)`

Update Directive

The update directive copies data between the memory for the encountering thread and the device. An update directive may appear in any data region, including an implicit data region.

C/C++

#pragma acc update [*clause* [,] *clause*]...

new-line

FORTRAN

!\$acc update [*clause* [,] *clause*]...

CLAUSES

self(*list*) or **host**(*list*)

Copies the data in *list* from the device to the encountering thread.

device(*list*)

Copies the data in *list* from the encountering thread to the device.

if_present

Issue no error when the data is not present on the device.

if(*condition*)

When the *condition* is zero or .FALSE., no data will be moved to or from the device.

async [(*expression*)]

The data movement will execute asynchronously with the encountering thread on the corresponding async queue.

wait [(*expression-list*)]

The data movement will not begin execution until all actions on the corresponding async queue(s) are complete.

Wait Directive

The wait directive causes the encountering thread to wait for completion of asynchronous device activities, or for asynchronous activities on one async queue to synchronize with one or more other async queues. With no expression, it will wait for all outstanding asynchronous regions or data movement.

C/C++

#pragma acc wait [(*expression-list*)] [*clause* [,] *clause*]...

new-line

FORTRAN

!\$acc wait [(*expression-list*)] [*clause* [,] *clause*...]

CLAUSE

async [(*expression*)]

Enqueue the wait operation on the associated device queue. The encountering thread may proceed without waiting.

Routine Directive

The **routine** directive tells the compiler to compile a procedure for the device and gives the execution context for calls to the procedure. Such a routine is called an device routine.

C/C++

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

#pragma acc routine(*name*) [*clause* [,] *clause*...] *new-line*

Without a name, the **routine** directive must be followed immediately by a function definition or prototype.

FORTRAN

!\$acc routine [*clause* [,] *clause*...]

!\$acc routine(*name*) [*clause* [,] *clause*...]

Without a name, the **routine** directive must appear in the specification part of a subroutine or function, or in the interface body of a subroutine or function in an interface block.

CLAUSE

gang

Specifies that the procedure may contain a gang-shared loop, therefore calls to this procedure must appear outside any gang-shared loop. All gangs must call the procedure.

worker

Specifies that the procedure may contain a worker-shared loop, therefore calls to this procedure must appear outside any worker-shared loop.

vector

Specifies that the procedure may contain a vector-shared loop, therefore calls to this procedure must appear outside any vector-shared loop.

seq

Specifies that the procedure has no device work-shared loops. A call to the procedure will be executed sequentially by the thread making the call.

bind(*name*)

Specifies an alternate procedure name to use when compiling or calling the procedure on the device.

bind(*string*)

Specifies a quoted string to use for the name when compiling or calling the procedure on the device.

device_type or **dtype**([*|*device-type-list*])

May be followed by any of the clauses below. Clauses following **device_type** will apply only when compiling for the given device type(s). Clauses following **device_type**(*) apply to all devices not named in another **device_type** clause.

nohost

Specifies that a host version of the procedure should not be compiled.

Global Data

C or C++ global, file static or extern objects, and Fortran module and common block variables and arrays that are used in device routines must appear in a **declare** directive in a **create**, **copyin**, **device_resident** or **link** clause.

Implicit Data Region

An implicit data region is created at the start of each procedure and ends after the last executable statement.

Declare Directive

A **declare** directive is used to specify that data is to be allocated in device memory for the duration of the implicit data region of the program or subprogram.

C/C++

```
#pragma acc declare [clause [,] clause]...]  
new-line
```

FORTRAN

```
!$acc declare [clause [,] clause]...
```

Data clauses are allowed.

OTHER CLAUSES

device_resident(*list*)

Specifies that the variables in *list* are to be allocated on the device for the duration of the implicit data region.

link(*list*)

For large global static data objects, specifies that a global link for each object in *list* is to be statically allocated on the device. Device memory for the object will be allocated when the object appears in a data clause, and the global link will be assigned.

Runtime Library Routines

Prototypes or interfaces for the runtime library routines, along with datatypes and enumeration types, are available as follows:

C/C++

```
#include "openacc.h"
```

FORTRAN

```
use openacc
```

C AND FORTRAN ROUTINES

In the following, **h_void*** is a **void*** pointer to host memory, and **d_void*** is a **void*** pointer to device memory.

acc_get_num_devices(devicetype)

Returns the number of devices of the specified type.

acc_set_device_type(devicetype)

Sets the device type to use for this host thread.

acc_get_device_type()

Returns the device type that is being used by this host thread.

acc_set_device_num(devicenum, devicetype)

Sets the device number to use for this host thread.

acc_get_device_num(devicetype)

Returns the device number that is being used by this host thread.

acc_init(devicetype)

Initializes the runtime system and sets the device type to use for this host thread.

acc_shutdown(devicetype)

Disconnects this host thread from the device.

acc_async_test(expression)

Returns nonzero or .TRUE. if all asynchronous activities with the given expression have been completed; otherwise returns zero or .FALSE.

acc_async_test_all()

Returns nonzero or .TRUE. if all asynchronous activities have been completed; otherwise returns zero or .FALSE.

acc_wait(expression)

Waits until all asynchronous activities associated with the given expression have been completed

acc_wait_all(expression)

Waits until all asynchronous activities have been completed.

acc_wait_async(expression, expression)

Enqueues a wait operation for the async queue associated with the first argument onto the async queue associated with the second argument.

acc_wait_all_async()

Enqueues a wait operation for the all async queues onto the async queue associated with the expression.

acc_get_default_async()

Returns the async queue used by default when no queue is specified in an async clause.

acc_set_default_async()

Sets the default async queue used by default when no queue is specified on an async clause.

acc_on_device(devicetype)

In a **parallel** or **kernels** region, this is used to take different execution paths depending on whether the program is running on a device or on the host.

acc_malloc(size_t)

Returns the address of memory allocated on the device device.

acc_free(d_void*)

Frees memory allocated by **acc_malloc**.

acc_map_data(h_void*, d_void*, size_t)

Creates a new data lifetime for the host address, using the device data in the device address, with the data length in bytes.

acc_unmap_data(h_void*)

Unmaps the data lifetime previously created for the host address by **acc_map_data**.

acc_deviceptr(h_void*)

Returns the device pointer associated with a host address. Returns NULL if the host address is not present on the device.

acc_hostptr(d_void*)

Returns the host pointer associated with a device address. Returns NULL if the device address is not associated with a host address.

acc_memcpy_to_device(d_void*, h_void*, size_t)**acc_memcpy_to_device_async(d_void*, h_void*, size_t, int)**

Copies data from the local thread memory to the device.

acc_memcpy_from_device(h_void*, d_void*, size_t)**acc_memcpy_from_device_async(h_void*, d_void*, size_t, int)**

Copies data from the device to the local thread memory.

acc_memcpy_device(d_void*, d_void*, size_t)**acc_memcpy_device_async(d_void*, d_void*, size_t, int)**

Copies data from one device memory location to another.

DATA MOVEMENT ROUTINES

The following data routines are called with C prototype:

```
routine( h_void*, size_t )
```

and in Fortran with interface:

```
subroutine routine( a )  
  type, dimension(:[:])... :: a  
subroutine routine( a, len )  
  type :: a  
  integer :: len
```

The async versions are called with C prototype:

```
routine_async( h_void*, size_t, int )
```

and in Fortran with interface:

```
subroutine routine_async( a, async )  
  type, dimension(:[:])... :: a  
  integer :: async  
subroutine routine( a, len, async )  
  type :: a  
  integer :: len, async
```

acc_copyin, acc_copyin_async

Acts like an **enter data** directive with a **copyin** clause. Tests if the data is present, and if not allocates memory on and copies data to the current device. Increments the dynamic reference count.

acc_create, acc_create_async

Acts like an **enter data** directive with a **create** clause. Tests if the data is present, and if not allocates memory on the current device. Increments the dynamic reference count.

acc_copyout, acc_copyout_async

Acts like an **exit data** directive with a **copyout** and no **finalize** clause. Decrements the dynamic reference count. If both reference counts are zero, copies data from and deallocates memory on the current device.

acc_copyout_finalize, acc_copyout_finalize_async

Acts like an **exit data** directive with a **copyout** and **finalize** clause. Zeros the dynamic reference count. If both reference counts are zero, copies data from and deallocates memory on the current device.

acc_delete, acc_delete_async

Acts like an **exit data** directive with a **delete** and no **finalize** clause. Decrements the dynamic reference count. If both reference counts are zero, deallocates memory on the current device.

acc_delete_finalize, acc_delete_finalize_async

Acts like an **exit data** directive with a **delete** and a **finalize** clause. Zeros the dynamic reference count. If both reference counts are zero, deallocates memory on the current device.

acc_update_device, acc_update_device_async

Acts like an **update** directive with a **device** clause. Updates the corresponding device memory from the host memory.

acc_update_self, acc_update_self_async

Acts like an **update** directive with a **self** clause. Updates the host memory from the corresponding device memory.

acc_is_present

Tests whether the specified host data is present on the device. Returns nonzero or .TRUE. if the data is fully present on the device.

Environment Variables

ACC_DEVICE_TYPE *device*

The variable specifies the device type to which to connect.

This can be overridden with a call to **acc_set_device_type**.

ACC_DEVICE_NUM *num*

The variable specifies the device number to which to connect.

This can be overridden with a call to **acc_set_device_num**.

Conditional Compilation

The `_OPENACC` preprocessor macro is defined to have value `yyyyymm` when compiled with OpenACC directives enabled. The version described here has value 201510.

More OpenACC resources available at
www.openacc.org

