



## **PGI Accelerator Programming Model for Fortran & C**

---

The Portland Group

Published: v1.3 November 2010

# Contents

1. Introduction .....	5
1.1 Scope .....	5
1.2 Glossary .....	5
1.3 Execution Model.....	7
1.4 Memory Model .....	7
1.5 Changes since version 1.2.....	8
1.6 Organization of this document .....	8
1.7 References .....	8
2. Directives.....	11
2.1 Directive Format .....	11
2.2 Conditional Compilation .....	12
2.3 Accelerator Region Directives .....	12
2.3.1 Accelerator Compute Region Directive .....	12
2.3.2 Accelerator Data Region Directive .....	14
2.3.3 data clauses .....	14
2.3.4 update clauses .....	16
2.4 Accelerator Loop Mapping Directives .....	17
2.4.1 loop scheduling clauses .....	17
2.4.2 independent clause .....	19
2.4.3 kernel clause .....	19
2.4.4 shortloop clause.....	19
2.4.5 private clause .....	19
2.4.6 cache clause.....	20
2.5 Combined Directives.....	20
2.6 Declarative Data Directives .....	20
2.6.1 copy declarative clause.....	22
2.6.2 copyin declarative clause .....	22
2.6.3 copyout declarative clause .....	22
2.6.4 local declarative clause .....	22
2.6.5 deviceptr declarative clause .....	22
2.6.6 device resident declarative clause .....	23
2.6.7 mirror declarative clause.....	23
2.6.8 reflected declarative clause .....	23
2.7 Executable Directives .....	23
2.7.1 update directive.....	23
2.7.2 device present directive .....	24
2.7.3 wait directive.....	24
3. Runtime Library Routines .....	27
3.1 Runtime Library Definitions .....	27
3.2 Runtime Library Routines .....	27
3.2.1 acc_get_num_devices .....	27
3.2.2 acc_set_device .....	28
3.2.3 acc_get_device.....	28
3.2.4 acc_set_device_num .....	29
3.2.5 acc_get_device_num.....	30

3.2.6 acc_async_test.....	30
3.2.7 acc_async_wait .....	31
3.2.8 acc_init.....	31
3.2.9 acc_shutdown .....	32
3.2.10 acc_on_device .....	32
4. Environment Variables.....	33
4.1 ACC_DEVICE .....	33
4.2 ACC_DEVICE_NUM.....	33
4.3 ACC_NOTIFY .....	33
5. Limitations .....	35



# The Portland Group

## 1. Introduction

This document describes a collection of compiler directives used to specify regions of code in Fortran and C programs that can be offloaded from a *host* CPU to an attached *accelerator*. The method outlined provides a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators. The directives extend the ISO/ANSI standard C and Fortran base languages in a way that allows a programmer to migrate applications incrementally to accelerator targets using standards-compliant Fortran or C.

The directives and programming model defined in this document allow programmers to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown. Rather, all of these details are implicit in the programming model and are managed by the PGI Fortran & C accelerator compilers. The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator region, guidance on mapping of loops onto an accelerator, and similar performance-related details.

### 1.1 Scope

This PGI Fortran & C accelerator programming model document covers only user-directed accelerator programming, where the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The bulk of a user's program, as well as regions containing constructs that are not supported on the targeted accelerator, will be executed on the host. This document does not describe features or limitations of the host programming environment as a whole; it is limited to specification of loops and regions of code to be offloaded to an accelerator.

This document does not describe automatic detection and offloading of regions of code to an accelerator by a compiler or other tool. This document does not describe targeting of accelerator regions to multiple accelerators attached to a single host. While future compilers may allow for automatic offloading, multiple accelerators of the same type, or multiple accelerators of different types, none of these features are addressed in this document.

### 1.2 Glossary

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

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

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

**Compute region** – a *region* defined by an Accelerator compute region directive. A compute region is a structured block containing loops which are compiled for the accelerator. A compute region may require device memory to be allocated and data to be copied from host to device upon region entry, and data to be copied from device to host memory and device

memory deallocated upon exit. Compute regions may not contain other compute regions or data regions.

**CUDA** – short for Compute Unified Device Architecture; the CUDA environment from NVIDIA is a C-like programming environment used to explicitly control and program an NVIDIA GPU.

**Data region** – a *region* defined by an Accelerator data region directive, or an implicit data region for a function or subroutine containing Accelerator directives. Data regions typically require device memory to be allocated and data to be copied from host to device memory upon entry, and data to be copied from device to host memory and device memory deallocated upon exit. Data regions may contain other data regions and compute regions.

**Device** – a general reference to any type of accelerator.

**Device memory** – memory attached to an accelerator, physically separate from the host memory.

**Directive** – in C, a `#pragma`, or in Fortran, a specially formatted comment statement, that is interpreted by a compiler to augment information about or specify the behavior of the program.

**DMA** – Direct Memory Access, a method to move data between physically separate memories; this is typically performed by a DMA engine, separate from the host CPU, that can access the host physical memory as well as an IO device or GPU physical memory.

**GPU** – a Graphics Processing Unit; one type of accelerator device.

**GPGPU** – General Purpose computation on Graphics Processing Units.

**Host** – the main CPU that in this context has an attached accelerator device. The host CPU controls the program regions and data loaded into and executed on the device.

**Loop trip count** – the number of times a particular loop executes.

**OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming environment that enables low-level general-purpose programming on GPUs and other accelerators.

**Private data** – with respect to an iterative loop, data which is used only during a particular loop iteration. With respect to a more general region of code, data which is used within the region but is not initialized prior to the region and is re-initialized prior to any use after the region.

**Region** – a structured block identified by the programmer or implicitly defined by the language. Certain actions may occur when program execution reaches the start and end of a region, such as device memory allocation or data movement between the host and device memory. Loops in a compute region are targeted for execution on the accelerator.

**Structured block** – in C, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.

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

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

### 1.3 Execution Model

The execution model targeted by the PGI accelerator compilers is host-directed execution with an attached accelerator device, for example a GPU. The bulk of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The device executes kernels, which may be as simple as a tightly-nested loop, or as complex as a subroutine, depending on the accelerator hardware. Even in accelerator-targeted regions, the host must orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the kernel code to the accelerator, passing kernel arguments, queueing the kernel, waiting for completion, transferring results back to the host, and deallocating memory. In most cases, the host can queue a sequence of kernels to be executed on the device, one after the other.

Most current GPUs support two levels of parallelism: an outer *doall* (fully parallel) loop level, and an inner *synchronous* (SIMD or vector) loop level. Each level can be multidimensional with 2 or 3 dimensions, but the domain must be strictly rectangular. The *synchronous* level may not be fully implemented with SIMD or vector operations, so explicit synchronization is supported and required across this level. No synchronization is supported between parallel threads across the *doall* level. The execution model on the device side exposes these two levels of parallelism and the programmer is required to understand the difference between, for example, a fully parallel loop and a loop that is vectorizable but requires synchronization across iterations. All fully parallel loops can be scheduled for either *doall* or *synchronous* parallel execution, but by definition SIMD vector loops that require synchronization can only be scheduled for synchronous parallel execution.

### 1.4 Memory Model

The most significant difference between a host-only program and a host+accelerator program is that the memory on the accelerator can be completely separate from host memory. This is the case on most current GPUs, for example. In this case, the host cannot read or write device memory directly because it is not mapped into the host's virtual memory space. All data movement between host memory and device memory must be performed by the host through runtime library calls that explicitly move data between the separate memories, typically using DMA. Similarly, it is not valid to assume the accelerator can read or write host memory, though this may be supported by accelerators in the future.

The concept of separate host and accelerator memories is very apparent in low-level accelerator programming models such as CUDA or OpenCL, in which data movement between the memories dominates user code. In the PGI accelerator programming model, data movement between the memories is implicit and managed by the compiler, with hints from the programmer in the form of directives. However, the programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and device memory determines the level of compute intensity required to effectively accelerate a given region of code
- The limited device memory size may prohibit offloading of regions of code that operate on very large amounts of data

On the accelerator side, current GPUs implement a weak memory model. In particular, they do not support memory coherence between threads unless those threads are parallel only at the *synchronous* level and the memory operations are separated by an explicit barrier. Otherwise,

if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, the hardware does not guarantee the results. While the results of running such a program might be inconsistent, it is not accurate to say that the results are incorrect. By definition, such programs are defined as being in error. While a compiler can detect some potential errors of this nature, it is nonetheless possible to write an accelerator region that produces inconsistent numerical results.

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

## 1.5 Changes since version 1.2

The `deviceptr` data clause has been added for C. The `reflected` data clause has now been defined for C as well as Fortran. A new form of the declarative data directive has been defined for use with C function prototypes.

The `device present` executable directive and `device resident` data clause have been added.

Asynchronous data updates and asynchronous compute regions have been added, with associated synchronization directive and runtime routines.

The `shortloop` loop scheduling clause has been redefined; it was originally required the programmer to ensure that the loop trip count was small enough that any device limits would not be exceeded. It is not defined to be guidance to the compiler that the loop trip count is likely to be relatively small, and that the compiler should optimize for that case.

## 1.6 Organization of this document

The rest of this document is organized as follows:

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

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

Chapter 4, *Environment Variables*, defines user-settable environment variables used to control behavior of accelerator-enabled programs at execution.

## 1.7 References

- ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part 1: Base Language*, Geneva, 2003 (Fortran 2003).
- *American National Standard Programming Language C*, ANSI X3.159-1989 (ANSI C).
- ISO/IEC 9899:1999, *Information Technology – Programming Languages – C*, Geneva, 1999 (C99).

- *PGI User's Guide*, The Portland Group, Release 11.0, December, 2010. Available online at <http://www.pgroup.com/doc/pgiug.pdf>.
- *PGI Tools Guide*, The Portland Group, Release 11.0, December, 2010. Available online at <http://www.pgroup.com/doc/pgitools.pdf>.
- *PGI Fortran Reference*, The Portland Group, Release 11.0, December, 2010. Available online at <http://www.pgroup.com/doc/pgifortref.pdf>.



## 2. Directives

This chapter describes the syntax and behavior of the PGI Accelerator directives. In C, Accelerator directives are specified using the `#pragma` mechanism provided by the standard. In Fortran, Accelerator directives are specified using special comments that are identified by a unique sentinel.

Compilers can ignore Accelerator directives if support is disabled or not provided. PGI compilers enable Accelerator directives with the `-ta` command line option.

### 2.1 Directive Format

In C, Accelerator directives are specified with the `#pragma` mechanism. The syntax of an Accelerator directive is:

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

Each directive starts with `#pragma acc`. The remainder of the directive follows the 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. An Accelerator directive applies to the immediately following structured block or loop.

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

```
!$acc directive-name [clause [,clause]...]
```

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 a 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. Comments may appear on the same line as the 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, directives are specified as one of

```
!$acc directive-name [clause [,clause]...]
```

```
c$acc directive-name [clause [,clause]...]
```

```
*$acc directive-name [clause [,clause]...]
```

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

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

Only one *directive-name* can be specified per directive. The order in which clauses appear is not significant, and clauses may be repeated unless otherwise specified. Some clauses have a *list* argument; a *list* is a comma-separated list of variable names, array names, or, in some cases, subarrays with subscript ranges.

## 2.2 Conditional Compilation

The `_ACCEL` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is the month designation of the version of the Accelerator directives supported by the implementation. This macro must be defined by a compiler only when Accelerator directives are enabled. The version described here is 201011.

## 2.3 Accelerator Region Directives

### 2.3.1 Accelerator Compute Region Directive

#### Summary

This directive defines a region of the program that should be compiled for execution on the accelerator device.

#### Syntax

In C, the syntax of the Accelerator compute region directive is

```
#pragma acc region [clause [, clause]...] new-line
                structured block
```

and in Fortran, the syntax is

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

where *clause* is one of the following:

```
if( condition )
async [( handle )]
copy( list )
copyin( list )
copyout( list )
local( list )
deviceptr( list )
update device( list )
update host( list )
```

#### Description

Loops within the structured block will be compiled into accelerator kernels. Data will be copied from the host memory to the device memory, as required, and result data will be copied back. Any computation that cannot be executed on the accelerator, perhaps because of limitations of the device, will be executed on the host. This may require data to move back and forth between the host and device.

At the end of the region, all results stored on the device that are needed on the host will be copied back to the host memory, and any device memory allocated for the compute region will be deallocated.

The data clauses are optional. They are described in Section 2.3.3. For each variable or array used in the compute region that does not appear in any data clause, the compiler will analyze all references to the variable or array and determine:

- For arrays, how much memory needs to be allocated in the device memory to hold the array;
- Whether the value in host memory needs to be copied to the device memory;
- Whether a value computed on the accelerator will be needed again on the host, and therefore needs to be copied back to the host memory.

When compiler analysis is unable to determine these items, it may fail to generate code for the accelerator; in that case, it should issue a message to notify the programmer why it failed. The data clauses can be used to augment or override this compiler analysis.

### **Restrictions**

- Accelerator compute regions may not be nested.
- A program may not branch into or out of an Accelerator compute region.
- A program must not depend on the order of evaluation of the clauses, or on any side effects of the evaluations.
- At most one `if` clause may appear. In Fortran, the condition must evaluate to a scalar logical value; in C, the condition must evaluate to a scalar integer value.

The `copy`, `copyin`, `copyout`, `local`, and `deviceptr` clauses are described in Section 2.3.3.

#### **2.3.1.1 if clause**

The `if` clause is optional; when there is no `if` clause, the compiler will generate code to execute as much of the compute region on the accelerator as possible.

When an `if` clause appears, the compiler will generate two copies of the compute region, one copy to execute on the accelerator and one copy to execute on the host. When the *condition* in the `if` clause evaluates to zero in C, or `.false.` in Fortran, the host copy will be executed. When the *condition* evaluates to nonzero in C, or `.true.` in Fortran, the accelerator copy will be executed.

#### **2.3.1.2 async clause**

The `async` clause is optional; when there is no `async` clause, the host process will wait until the accelerator region is complete before executing any of the code that follows the region. When there is an `async` clause, the accelerator region will be processed by an auxiliary thread while the host process continues with the code following the region. The

auxiliary thread will manage the data movement and execution of the computation on the accelerator.

If the `async` clause has an argument, that argument must be the name of an integer variable (`int` for C, `integer(4)` for Fortran). The variable may be used in a `wait` directive or various runtime routines to make the host process wait for completion of the region.

Two asynchronous activities will be processed by the auxiliary thread in the order in which the host process encounters them.

## 2.3.2 Accelerator Data Region Directive

### Summary

This directive defines data, typically arrays, that should be allocated in the device memory for the duration of the data region, whether data should be copied from the host to the device memory upon region entry, and copied from the device to host memory upon region exit.

### Syntax

In C, the syntax of the Accelerator data region directive is

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

and in Fortran, the syntax is

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

where *clause* is one of the following:

```
copy( list )  
copyin( list )  
copyout( list )  
local( list )  
deviceptr( list )  
mirror( list )  
update device( list )  
update host( list )
```

### Description

Data will be allocated in the device memory and copied from the host memory to the device, or copied back, as required. The clauses are described in Sections 2.3.3 and 2.3.4.

### 2.3.3 data clauses

The list argument to each data clause is a comma-separated collection of variable names, array names, or subarray specifications. In all cases, the compiler will allocate and manage a copy of the variable or array in device memory, creating a visible device copy of that variable

or array. In C, a subarray is an array name followed by a range specification in brackets, such as

```
arr[2:high][low:100]
```

In Fortran, a subarray is an array name followed by a comma-separated list of range specifications in parentheses, such as

```
arr(2:high,low:100)
```

If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if known, are used. Using an array name in a data clause on a compute region directive without bounds tells the compiler to analyze the references to the array to determine what bounds to use. Thus, every array reference is equivalent to some subarray of that array.

### Restrictions

- A variable, array, or subarray may appear at most once in all data clauses for a compute or data region.
- Only one subarray of an array may appear in all data clauses for a region.
- If a variable, array, or subarray appears in a data clause for a region, the same variable, array, or any subarray of the same array may not appear in a data clause for any enclosed region.
- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C, a missing lower bound is assumed to be zero. A missing upper bound for a dynamically allocated array must be specified.
- If a subarray is specified in a data clause, the compiler may choose to allocate memory for only that subarray on the accelerator.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- The `mirror` clause is valid only in Fortran. The list argument to the mirror clause is a comma-separated list of array names. The arrays may be explicit shape, assumed shape, or allocatable; pointer arrays are not allowed.
- In Fortran, pointer arrays may be specified, but pointer association is not preserved in the device memory.

#### 2.3.3.1 copy clause

The `copy` clause is used to declare that the variables, arrays or subarrays in the *list* have values in the host memory that need to be copied to the device memory, and are assigned values on the accelerator that need to be copied back to the host. If a subarray is specified, then only that subarray of the array needs to be copied. The data is copied to the device memory upon entry to the region, and data copied back to the host memory upon exit from the region.

#### 2.3.3.2 copyin clause

The `copyin` clause is used to declare that the variables, arrays or subarrays in the *list* have values in the host memory that need to be copied to the device memory. If a subarray is specified, then only that subarray of the array needs to be copied. If a variable, array or subarray appears in a `copyin`, the clause implies that the data need not be copied back from

the device memory to the host memory, even if those values were changed on the accelerator. The data is copied to the device memory upon entry to the region.

### **2.3.3.3 copyout clause**

The `copyout` clause is used to declare that the variables, arrays or subarrays in the *list* are assigned or contain values in the device memory that need to be copied back to the host memory at the end of the accelerator region. If a subarray is specified, then only that subarray of the array needs to be copied. If a variable, array or subarray appears in a `copyout`, the clause implies that the data need not be copied to the device memory from the host memory, even if those values are used on the accelerator. The data is copied back to the host memory upon exit from the region.

### **2.3.3.4 local clause**

The `local` clause is used to declare that the variables, arrays or subarrays in the *list* need to be allocated in the device memory, but the values in the host memory are not needed on the accelerator, and any values computed and assigned on the accelerator are not needed on the host.

### **2.3.3.5 deviceptr clause**

The `deviceptr` clause is valid only in C. The `deviceptr` clause is used to declare that the pointers in the *list* are device pointers, so the data need not be allocated or moved between the host and device for this pointer.

### **2.3.3.6 mirror clause**

The `mirror` clause is only valid in Fortran on the Accelerator data region directive. The `mirror` clause is used to declare that the arrays in the *list* need to mirror the allocation state of the host array within the region. If the host array is allocated upon region entry, the device copy of the array will be allocated at region entry to the same size. If the host array is not allocated, the device copy will be initialized to an unallocated state. If the host array is allocated or deallocated within the region, the device copy will be allocated to the same size, or deallocated, at the same point in the region. At region exit, the device copy will be automatically deallocated, if it is still allocated at that point.

## **2.3.4 update clauses**

The *list* argument to each update clause is a comma-separated collection of variable names, array names, or subarray specifications. All variables or arrays that appear in the *list* argument of an update clause must have a visible device copy outside the compute or data region. The effect of an `update device` clause is to copy the variables, arrays, or subarrays in the *list* argument from host memory to the visible device copy of the variables, arrays or subarrays in device memory, before beginning execution of the compute or data region. The effect of an `update host` clause is to copy the visible device copies of the variables, arrays, or subarrays in the *list* argument to the associated host memory locations, after completion of the compute or data region. Multiple subarrays of the same array may appear in update clauses for the same region, potentially causing updates of different subarrays in each direction.

## 2.4 Accelerator Loop Mapping Directives

### Summary

The Accelerator loop mapping directive applies to a loop which must appear on the following line. It can describe what type of parallelism to use to execute the loop and declare loop-private variables and arrays.

### Syntax

In C, the syntax of the Accelerator loop mapping directive is

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

In Fortran, the syntax of the Accelerator loop mapping directive is

```
!$acc do [clause [,clause]...]  
    do loop
```

where *clause* is one of the following:

```
host [(width)]  
parallel [(width)]  
seq [(width)]  
vector [(width)]  
unroll (factor)  
independent  
kernel  
private( list )  
cache( list )
```

#### 2.4.1 loop scheduling clauses

The loop scheduling clauses are optional. For each loop without a scheduling clause, the compiler will determine an appropriate schedule automatically.

The loop schedule clauses tell the compiler about loop level parallelism and how to map the parallelism onto the accelerator parallelism. In some cases, there is a limit on the trip count of a parallel loop on the accelerator. For instance, some accelerators have a limit on the maximum length of a vector loop. In such cases, the compiler will strip-mine the loop, so that one of the loops has a maximum trip count that satisfies the limit. For instance, if the maximum vector length is 256, the compiler will compile a vector loop like:

```
!$acc do vector  
    do i = 1, n
```

into the following pair of loops, using strip-mining:

```
do is = 1, n, 256  
    !$acc do vector  
        do i = is, max(is+255, n)
```

The compiler will then choose an appropriate schedule for the outer, strip loop.

If more than one scheduling clause appears on the loop directive, the compiler will strip-mine the loop to get at least that many nested loops, applying one loop scheduling clause to each level. If a loop scheduling clause has a width argument, the compiler will strip-mine the loop

to that width, applying the scheduling clause to the outer strip or inner element loop, and then determine the appropriate schedule for the other loop. In an example like:

```
!$acc do host(16), parallel
  do i = 1,n
```

the compiler will strip-mine the loop to 16 host iterations, with the parallel clause applying to the inner loop, as follows:

```
ns = ceil(n/16)
!$acc do host
  do is = 1, n, ns
    !$acc do parallel
      do i = is, min(n, is+ns-1)
```

#### **2.4.1.1 host clause**

The `host` clause tells the compiler to execute this loop sequentially on the host processor. There is no maximum number of iterations on a host schedule. If a width argument appears, the compiler strip mines the loop with width iterations in each strip.

#### **2.4.1.2 parallel clause**

The `parallel` clause tells the compiler to execute this loop in parallel mode (*doall* parallelism) on the accelerator. There may be an accelerator target-specific limit on the number of iterations in a parallel loop; in that case, if there is no width argument, or the value of the width expression is greater than the limit, the compiler will enforce the limit. If there is a width argument or a limit on the number of iterations in a parallel loop, then only that many iterations will run in parallel at a time.

#### **2.4.1.3 seq clause**

The `seq` clause tells the compiler to execute this loop sequentially on the accelerator. There is no maximum number of iterations for a seq schedule. If a width argument appears, the compiler strip mines the loop with width iterations in each strip.

#### **2.4.1.4 vector clause**

The `vector` clause tells the compiler to execute this loop in vector mode (*synchronous* parallelism) on the accelerator. The width argument determines how many iterations are in a vector. There may be an accelerator target-specific limit on the number of iterations in a vector loop; in that case, if there is no width argument, or the value of the width expression is greater than the limit, the compiler will enforce the limit through strip-mining.

#### **2.4.1.5 unroll clause**

The `unroll` clause tells the compiler to unroll iterations for sequential execution on the accelerator. The unroll clause applies to the most recent `parallel`, `seq` or `vector` clause to unroll the parallel, sequential, or vector iterations of the loop; the unroll clause may appear multiple times, once for each `parallel`, `seq` or `vector` clause. The unroll factor argument must be a compile time positive constant integer.

### **Restrictions**

- If two or more loop scheduling clauses (excepting the `unroll` clause) appear on a single loop mapping directive, all but one must have a width argument.
- An `unroll` clause must always specify an unroll factor.

- Some implementations or targets may require the width expression for the vector clause to be a compile-time constant.
- Some implementations or targets may require the width expression for the vector or parallel clauses to be a power of two, or a multiple of some power of two. If so, the behavior when the restriction is violated is implementation-defined.

#### **2.4.2 independent clause**

The `independent` clause tells the compiler that the iterations of this loop are data-independent of each other. This allows the compiler to generate code to execute the iterations in parallel, without synchronization.

#### **Restrictions**

- It is a programming error to use the `independent` clause if any iteration writes to a variable or array element that any other iteration also writes or reads.

#### **2.4.3 kernel clause**

The `kernel` clause tells the compiler that the body of this loop is to be the body of the computational kernel. Any loops contained within the kernel loop will be executed sequentially on the accelerator.

#### **Restrictions**

- Loop mapping directives must not appear on any loop contained within the `kernel` loop.

#### **2.4.4 shortloop clause**

The `shortloop` clause informs the compiler that the loop trip count is likely to be small relative to other nested loops. The compiler will use this the `shortloop` information when automatically choosing a loop schedule.

#### **2.4.5 private clause**

The `private` clause is used to declare that the variables, arrays, or subarrays in the `list` need to be allocated in the device memory with one copy for each iteration of the loop. Moreover, any value of the variable or array used in the loop must have been computed and assigned in that iteration of the loop, and the values computed and assigned in any iteration are not needed after completion of the loop. Using an array name without bounds tells the compiler to analyze the references to the array to determine what bounds to use. If the lower or upper bounds are missing, the declared or allocated bounds, if known, are used.

#### **Restrictions**

- A variable, array or subarray may only appear once in any `private` clause for a region.
- Only one subarray for an array may appear in any `private` clause for a region.
- If a subarray appears in a `private` clause, then the compiler only needs to allocate that subarray in the device memory.
- The compiler may pad dimensions of allocated arrays or subarrays to improve memory alignment and program performance.
- If a subarray appears in a `private` clause, it is an error to refer to any element of the array in the loop outside the bounds of the subarray.

- It is an error to refer to a variable or any element of an array or subarray that appears in a `private` clause and that has not been assigned in this iteration of the loop.
- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C, a missing lower bound is assumed to be zero. A missing upper bound for a dynamically allocated array must be specified.

#### 2.4.6 cache clause.

The `cache` clause is used to give a hint to the compiler to try to move the variables, arrays, or subarrays in the *list* to the highest level of the memory hierarchy. Many accelerators have a software-managed fast cache memory, and the `cache` clause can help the compiler choose what data to keep in that fast memory for the duration of the loop. The compiler is not required to store all or even any of the data items in the cache memory.

## 2.5 Combined Directives

### Summary

The combined Accelerator compute region and loop mapping directive is a shortcut for specifying a loop directive nested immediately inside an accelerator compute region. The meaning is identical to explicitly specifying a compute region containing a loop directive. Any clause that is allowed on a compute region directive or a loop directive is allowed on a combined directive.

### Syntax

In C, the syntax of the combined Accelerator `region` and loop directive is:

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

In Fortran the syntax of the combined Accelerator `region` and loop directive is:

```
!$acc region do [clause [, clause]...]
    do loop
```

The associated region is the body of the loop which must immediately follow the directive. Any of the region or loop clauses may appear.

### Restrictions

- This combined Accelerator `region` and loop directive may not appear within the body of another accelerator compute region.
- The restrictions for the `region` directive and the loop directive apply.

## 2.6 Declarative Data Directives

### Summary

Declarative data directives are used in the declaration section of a Fortran subroutine, function, or module, or just following an array declaration in C. They specify that an array or arrays are to be allocated in the 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 the host to the device memory upon entry to the implicit data region, and from the device to the host memory upon exit from the implicit data region. These directives create a visible device copy of the variable or array.

### Syntax

In C, the syntax of the declarative data directive is:

```
#pragma acc declclause [, declclause]... new-line
```

or

```
#pragma acc function(name) declclause [, declclause]... new-line
```

In Fortran the syntax of the declarative data directive is:

```
!$acc declclause [, declclause]...
```

where *declclause* is one of the following:

```
copy( list )  
copyin( list )  
copyout( list )  
local( list )  
deviceptr( list )  
device resident( list )  
mirror( list )  
reflected( list )
```

The associated region is the implicit region associated with the function, subroutine, or program in which the directive appears. If the directive appears in a Fortran MODULE subprogram, the associated region is the implicit region for the whole program.

### Restrictions

- A variable or array may appear at most once in all declarative data clauses for a function, subroutine, program, or module.
- Subarrays are not allowed in declarative data clauses.
- If a variable or array appears in a declarative data clause, the same variable or array may not appear in a data clause for any region where the declaration of the variable is visible.
- In Fortran, assumed-size dummy arrays may not appear in declarative data clauses.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- The `mirror` clause is valid only in Fortran.

- The list argument to the `mirror` clause is a comma-separated list of array names. The arrays may be explicit shape, assumed shape, or allocatable; pointer arrays are not allowed.
- The list argument to the `reflected` clause is a comma-separated list of dummy argument array names. The arrays may be explicit shape, assumed shape, or allocatable.
- The `mirror` clause may be used for Fortran allocatable arrays in Module subprograms. The `copy`, `copyin`, `copyout`, `local`, and `reflected` clauses may not be used in Module subprograms.
- In Fortran, pointer arrays may be specified, but pointer association is not preserved in the device memory.
- In Fortran, the directive may appear in an interface block, to expose the `reflected` clause to the caller.
- In C, the second form of the directive may only appear after a prototype of the function; it is used to expose the `reflected` clause to the caller.

### **2.6.1 copy declarative clause**

The `copy` declarative clause is used to declare that the variables or arrays in the *list* are to be allocated in the device memory, and that the values in host memory need to be copied to the device memory upon entry to the implicit region associated with this directive, and the values from the device memory should be copied back to the host memory upon region exit.

### **2.6.2 copyin declarative clause**

The `copyin` declarative clause is used to declare that the variables or arrays in the *list* are to be allocated in the device memory, and that the values in host memory need to be copied to the device memory upon entry to the implicit region associated with this directive. The clause implies that the data need not be copied back from the device memory to the host memory, even if those values were changed on the accelerator.

### **2.6.3 copyout declarative clause**

The `copyout` declarative clause is used to declare that the variables or arrays in the *list* are to be allocated in the device memory, and that the values in device memory need to be copied back to the host memory upon exit from the implicit region associated with this directive. The clause implies that the values need not be copied to the device memory from the host memory at entry to the region.

### **2.6.4 local declarative clause**

The `local` declarative clause is used to declare that the variables or arrays in the *list* are to be allocated in the device memory. The clause implies that the values in the host memory are not needed on the accelerator, and any values computed and assigned on the accelerator are not needed on the host.

### **2.6.5 deviceptr declarative clause**

The `deviceptr` declarative clause is only valid in C. The clause is used to declare that the pointers in the *list* are device pointers, so the data need not be allocated or moved between the host and device for any use of the pointers in accelerator compute regions.

### 2.6.6 device resident declarative clause

The `device resident` declarative clause is used to declare that the variables or arrays in the list are to be allocated on the device only. In Fortran, the items in the list may be allocatable, in which case the allocation will be done only on the device.

### 2.6.7 mirror declarative clause

The `mirror` declarative clause is only valid in Fortran. The `mirror` clause is used to declare that device copies of the arrays in the *list* should mirror the allocation state of the host array within the implicit region associated with this directive. If the host array is allocated upon region entry, the device copy of the array will be allocated at region entry to the same size. If the host array is not allocated, the device copy will be initialized to an unallocated state. If the host array is allocated or deallocated within the region, the device copy will be allocated to the same size, or deallocated, at the same point in the region. At region exit, if the device copy will be automatically deallocated, if it is allocated at that point. When used in a Fortran module subprogram, the associated region is the implicit region for the whole program.

### 2.6.8 reflected declarative clause

The `reflected` clause is used to declare that the actual argument arrays bound to the dummy argument arrays in the *list* must have a visible device copy at the call site. If the `reflected` declarative clause is used, the caller must have an explicit interface to this subprogram. If a Fortran interface block is used to describe the explicit interface, a matching `reflected` directive must appear in the interface block. In C, if the function is called from another file, a function prototype must appear in that source file with a directive specifying a matching `reflected` clause. The device copy of the array used within the subroutine or function will be the device copy that is visible at the call site. See also the `device present` executable directive.

## 2.7 Executable Directives

### 2.7.1 update directive

#### Summary

The `update` directive is used within an explicit or implicit data region to update all or part of a host memory array with values from the corresponding array in device memory, or to update all or part of a device memory array with values from the corresponding array in host memory.

#### Syntax

In C, the syntax of the `update` directive is:

```
#pragma acc update updateclause [, updateclause]... new-line
```

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

```
!$acc update updateclause [, updateclause]...
```

where *updateclause* is one of the following:

```
host ( list )  
device ( list )  
async [ ( handle ) ]
```

The list argument to an update clause is a comma-separated collection of variable names, array names, or subarray specifications. Multiple subarrays of the same array may appear in a list. The effect of an update clause is to copy data from the device memory to the host memory for `update host`, and from host memory to device memory for `update device`. The updates are done in the order in which they appear on the directive.

### 2.7.1.1 `async` clause

#### Summary

The `async` clause is optional; when there is no `async` clause, the host process will wait until the updates are complete before executing any of the code that follows the `update` directive. When there is an `async` clause, the updates will be processed by an auxiliary thread while the host process continues with the code following the directive.

If the `async` clause has an argument, that argument must be the name of an integer variable (`int` for C, `integer(4)` for Fortran). The variable may be used in a `wait` directive or various runtime routines to make the host process wait for completion of the update.

Two asynchronous activities will be processed by the auxiliary thread in the order in which the host process encounters them.

#### Restrictions

- The update directive is executable. It must not appear in place of the statement following an *if*, *while*, *do*, *switch*, or *label* in C, or in place of the statement following a logical *if* in Fortran.
- A variable or array which appears in the list of an `update` directive must have a visible device copy.

### 2.7.2 `device present` directive

#### Summary

The `device present` directive is used to find an existing device copy of arrays.

#### Syntax

In C, the syntax of the `device present` directive is:

```
#pragma acc device present( list ) new-line
```

In Fortran the syntax of the `device present data` directive is:

```
!$acc device present( list )
```

The arrays in the list must have appeared in a data clause for some data region surrounding the call to the procedure in which this directive appears.

### 2.7.3 `wait` directive

#### Summary

The `wait` directive causes the program to wait for completion of an asynchronous activity, such as an accelerator region or `update` directive.

#### Syntax

In C, the syntax of the `wait` directive is:

```
#pragma acc wait( handle ) new-line
```

In Fortran the syntax of the `wait` directive is:

```
!$acc wait( handle )
```

The `handle` argument, if specified, must be the name of an integer variable (`int` for C, `integer(4)` for Fortran), which has been initialized to zero, or which appeared in an `async` clause. If the value is zero, the host process will not wait for any asynchronous activity. If the variable appeared in an `async` clause (or several `async` clauses), the host process will wait until the latest such asynchronous activity completes. If no `handle` argument appears, the host process will wait for all asynchronous activities to complete.



## 3. Runtime Library Routines

This chapter describes the PGI Accelerator runtime library routines that are available for use by programmers. This chapter has two sections:

- Runtime library definitions
- Runtime library routines

### Restrictions

- In Fortran, none of the Accelerator runtime library routines may be called from a `PURE` or `ELEMENTAL` procedure.

### 3.1 Runtime Library Definitions

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

- The prototypes of all routines in the chapter.
- Any datatypes used in those prototypes, including an enumeration type to describe types of accelerators.

In Fortran, interface declarations are provided in a Fortran include file named `accel_lib.h` and in a Fortran module named `accel_lib`. These files define:

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

Many of the routines accept or return a value corresponding to the type of accelerator device. In C, the datatype used for device type values is `acc_device_t`; in Fortran, the corresponding datatype is `integer(kind=acc_device_kind)`. The possible values for device type are implementation specific, and are listed in the C include file `accel.h`, the Fortran include file `accel_lib.h` and the Fortran module `accel_lib`. Three values are always supported: `acc_device_none`, `acc_device_default`, and `acc_device_host`. For other values, look at the appropriate files included with the implementation, or read the documentation for the implementation. The value `acc_device_default` will never be returned by any function; its use as an argument will tell the runtime library to select a default device type.

### 3.2 Runtime Library Routines

#### 3.2.1 `acc_get_num_devices`

##### Summary

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

## Format

C:

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

Fortran:

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

## Description

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

### 3.2.2 `acc_set_device`

#### Summary

The `acc_set_device` routine tells the runtime which type of device to use when executing an accelerator compute region. This is useful when the program has been compiled to use more than one type of accelerator.

## Format

C:

```
void acc_set_device ( acc_device_t );
```

Fortran:

```
subroutine acc_set_device ( devicetype )  
integer(acc_device_kind) devicetype
```

## Description

The `acc_set_device` routine tells the runtime which type of device to use among those. To be effective, this routine should be called before any accelerator data or compute regions have been entered, or after an `acc_shutdown` call.

## Restrictions

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

### 3.2.3 `acc_get_device`

#### Summary

The `acc_get_device` routine tells the program what type of device will be used to run the next accelerator region, if one has been selected. This is useful when the program has been compiled to use more than one type of accelerator.

## Format

C:

```
acc_device_t acc_get_device ( void );
```

Fortran:

```
function acc_get_device ()  
integer(acc_device_kind) acc_get_device
```

## Description

The `acc_get_device` routine returns a value to tell the program what type of device will be used to run the next accelerator region, if one has been selected. The device type may have been selected by the program with an `acc_set_device` call, with an environment variable, or by the default behavior of the program. This is only effective for accelerator regions that were compiled to run on more than one type of device.

## Restrictions

- This routine may not be called during execution of an accelerator compute region.
- If the device type has not yet been selected, the value `acc_device_none` will be returned.

### 3.2.4 `acc_set_device_num`

#### Summary

The `acc_set_device_num` routine tells the runtime which device to use when executing an accelerator region.

## Format

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
```

## Description

The `acc_set_device_num` routine tells the runtime which device to use among those attached of the given type. If the value of `devicenum` is zero, the runtime will revert to its default behavior, which is implementation-defined. If the value of the second argument is zero, the selected device number will be used for all attached accelerator types.

## Restrictions

- This routine may not be called during execution of an accelerator region.
- If the value of `devicenum` is greater than the value returned by `acc_get_num_devices` for that device type, the behavior is implementation-defined.
- Calling `acc_set_device_num` implies a call to `acc_set_device` with that device type argument.

### 3.2.5 acc\_get\_device\_num

#### Summary

The `acc_get_device_num` routine returns the device number of the specified device type that will be used to run the next accelerator region.

#### Format

C:

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

Fortran:

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

#### Description

The `acc_get_device_num` routine returns an integer corresponding to the device number of the specified device type that will be used to execute the next accelerator region.

#### Restrictions

- This routine may not be called during execution of an accelerator region.

### 3.2.6 acc\_async\_test

#### Summary

The `acc_async_test` routine tests for completion of all associated asynchronous activities.

#### Format

C:

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

Fortran:

```
logical function acc_async_test( handle )  
integer(acc_handle_kind) handle
```

#### Description

The `handle` argument must be the name of an integer variable which has been initialized to zero, or which appeared in an `async` clause. If the value is zero, the `acc_async_test` routine will return immediately with a nonzero value or `.true`. If the variable appeared in an `async` clause (or several `async` clauses), and all such asynchronous activities have completed, the `acc_async_test` will return with a nonzero value or `.true`. If some such asynchronous activities have not completed, the `acc_async_test` will return with a zero value or `.false`.

#### Restrictions

- This routine may not be called during execution of an accelerator region.

### 3.2.7 `acc_async_wait`

#### Summary

The `acc_async_wait` routine waits for completion of all associated asynchronous activities.

#### Format

C:

```
void acc_async_wait( int );
```

Fortran:

```
subroutine acc_async_wait( handle )  
integer(acc_handle_kind) handle
```

#### Description

The `handle` argument must be the name of an integer variable which has been initialized to zero, or which appeared in an `async` clause. If the value is zero, the `acc_async_wait` routine will return immediately. If the variable appeared in an `async` clause (or several `async` clauses), the `acc_async_wait` routine will not return until the latest such asynchronous activity has completed. If the value is negative one, the routine will not return until all asynchronous activities have completed.

#### Restrictions

- This routine may not be called during execution of an accelerator region.

### 3.2.8 `acc_init`

#### Summary

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

#### Format

C:

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

Fortran:

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

#### Description

The `acc_init` routine also calls `acc_set_device`. To be effective, this routine should be called before any accelerator regions have been entered, or after an `acc_shutdown` call.

#### Restrictions

- This routine may not be called during execution of an accelerator region.
- If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.

- 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.
- If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.

### 3.2.9 `acc_shutdown`

#### Summary

The `acc_shutdown` routine tells the runtime to shut down the connection to the given accelerator device, and free up any runtime resources. This may be used to connect to a different device, if the program was built in a way to run on different device types.

#### Format

C:

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

Fortran:

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

#### Description

The `acc_shutdown` routine disconnects the program from the accelerator device.

#### Restrictions

- This routine may not be called during execution of an accelerator region.

### 3.2.10 `acc_on_device`

#### Summary

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

#### Format

C:

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

Fortran:

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

#### Description

The `acc_on_device` routine may be used to execute different paths depending on whether the code is running on the host or on some accelerator. If the `acc_on_device` routine has a compile-time constant argument, it evaluates at compile time to a constant. The argument must be one of the defined accelerator types. If the argument is `acc_device_host`, then outside of an accelerator compute region, or in an accelerator compute region that is compiled for the host processor, this routine will evaluate to nonzero for C, and `.true.` for Fortran; otherwise, it will evaluate to zero for C, and `.false.` for Fortran.

## 4. Environment Variables

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

### 4.1 ACC\_DEVICE

The `ACC_DEVICE` environment variable controls the default device type to use when executing accelerator regions, if the program has been compiled to use more than one different type of device. The value of this environment variable is implementation-defined. See the release notes for currently-supported values of this environment variable.

Example:

```
setenv ACC_DEVICE NVIDIA
export ACC_DEVICE=NVIDIA
```

### 4.2 ACC\_DEVICE\_NUM

The `ACC_DEVICE_NUM` environment variable controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices attached to the host. If the value is zero, the implementation-defined default is used. If the value is greater than the number of devices attached, the behavior is implementation-defined.

Example:

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

### 4.3 ACC\_NOTIFY

The `ACC_NOTIFY` environment variable can be used to print a short message to the standard output when a kernel is executed on an accelerator. The value of this environment variable must be a nonnegative integer. If the value is zero, no message is printed (the default behavior). If the value is nonzero, a one-line message is printed whenever an accelerator kernel is executed.

Example:

```
setenv ACC_NOTIFY 1
export ACC_NOTIFY=1
```



## 5. Limitations

This chapter contains known limitations and restrictions in the PGI Accelerator Programming Model for Fortran and C.

- **Loop nests to be offloaded to the GPU accelerator must be rectangular.** In particular, triangular loops or loops where the loop bound(s) of one loop are computed within another loop in the nest are not supported. Following is an example of a triangular loop:

```
for (j=0; j<n; j++)
  for (i=0; i<j; i++)
    <some code>
```

- **Pointers used to access arrays in loops that are to be offloaded to the GPU accelerator must be declared with the C99 'restrict' attribute.**

Alternatively, `safe_ptr` pragmas can be used or the whole file containing the loop can be compiled with the `-Msafe_ptr` option; however, these approaches can have unintended side effects.

- **At least some of the loops to be offloaded must be fully data parallel with no synchronization or dependences across iterations;** these loops enable distribution of work across the multi-processors in an NVIDIA GPU. One or more loops in the nest can be vectorizable loops that require some synchronization. For example, reductions are OK in many cases; these loops can be vectorized across multiple processors within a multiprocessor in an NVIDIA GPU. One or more loops in the nest can be sequential, but these loops are executed serially within a thread processor - e.g. as the inner loop(s).
- **Computed array indices should be avoided.** Such expressions within the loop nest result in the compiler detecting dependences that prevent parallelization and vectorization of loops. In future releases of the PGI Accelerator compiler, the 'independent' clause will allow the programmer to assert the independence of such loops and enable parallelization.
- **Function calls are not currently allowed within loops to be offloaded to a GPU accelerator.** In some cases, the compiler may be able to inline functions using the `-Minline` flag. However, it is recommended that you avoid calls within accelerator regions by manually inlining wherever possible. Improved automatic inlining will be supported in future versions of the PGI Accelerator compilers.
- **Loops that compute on structs can be offloaded, but those that operate on nested structs cannot.** This is a limitation of the current release of the compiler, not a limitation of the programming model.
- **Pointer arithmetic is not allowed within loops to be offloaded to the GPU accelerator.**

PGF95 and PGF90 are trademarks and PGI, PGI CDK, PGHPF, PGF77, PGCC, PGC++, PGI Visual Fortran, PVF, Cluster Development Kit, PGPROF, PGDBG and The Portland Group are registered trademarks of The Portland Group, Incorporated, a wholly-owned subsidiary of STMicroelectronics, Inc.

All other marks are the property of their respective owners.

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

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

© 2009-2010 The Portland Group, Incorporated. All rights reserved.