## Contents

1. Introduction .................................................................................................................. 1  
   1.1 Scope ......................................................................................................................... 1  
   1.2 Glossary .................................................................................................................... 1  
   1.3 Execution Model ....................................................................................................... 2  
   1.4 Memory Model ......................................................................................................... 3  
   1.5 Organization of this document .................................................................................. 3  
   1.6 References ................................................................................................................ 4  

2. Directives ...................................................................................................................... 5  
   2.1 Directive Format .......................................................................................................... 5  
   2.2 Conditional Compilation ............................................................................................ 6  
   2.3 Accelerator Region Directive .................................................................................... 6  
   2.4 Accelerator Loop Mapping Directives ....................................................................... 9  
   2.5 Combined Directives ................................................................................................. 12  

3. Runtime Library Routines ............................................................................................. 14  
   3.1 Runtime Library Definitions .................................................................................... 14  
   3.2 Runtime Library Routines ....................................................................................... 14  

4. Environment Variables .................................................................................................. 16  
   4.1 ACC_DEVICE_NUM ..................................................................................................... 16  

5. Installation and Supported Platforms .......................................................................... 17  

6. Restrictions & Limitations ............................................................................................ 18  
   6.1 Supported Directives ................................................................................................. 18  
   6.2 Programming Considerations ................................................................................. 18  
   6.3 Hardware Limitations for NVIDIA GPUs ................................................................. 18  

7. Basic Compiler Usage and Examples .......................................................................... 19  
   7.1 Using the PGI Accelerator Compilers ....................................................................... 19  
   7.2 A Simple Example .................................................................................................... 19  
   7.3 Matrix Multiplication ............................................................................................... 20  

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 does allow 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 compilers technology preview 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 cover automatic detection and offloading of regions of code to an accelerator by a compiler or other tool. This document does not cover targeting of accelerator regions to multiple accelerators attached to a single host. While future versions of the PGI compilers may allow for automatic offloading, multiple accelerators of the same type, or multiple accelerators of different types, none of these features are currently supported.

The PGI 8.0 Fortran & C Accelerator compilers technology preview is available only on x64 processor-based hosts running 64-bit Linux, and targets only a limited number and type of x64+GPU platforms. All examples included in this document are developed and presented on such a platform. For a list of supported GPUs, see Chapter 5, Installation and Supported Platforms.

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.

**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.

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

**Device memory** – memory attached to an accelerator which is 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.

**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 proposed standard C-like programming environment similar to CUDA that enables portable 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.

**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.

### 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 accelerator memory by reference because it is not mapped into the virtual memory space of the host. All data movement between host memory and accelerator memory must be performed by the host through runtime library calls that explicitly move data between the separate memories. 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, but the programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and accelerator memory determines the level of compute intensity required to effectively accelerate a given region of code
- Limited size of accelerator memory 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 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.

Chapter 5, *Installation and Supported Platforms*, describes how to install and enable the PGI Fortran and C accelerator compilers, software and hardware co-requirements, and supported platforms.

Chapter 6, *Restrictions and Limitations*, describes limitations in the PGI 8.0 implementation of the PGI accelerator programming model.

Chapter 7, *Basic Compiler Usage and Examples*, provides a basic outline of how to use the PGI Fortran and C accelerator compilers, relevant compiler options, and several examples of how to build and run accelerator-enabled programs.

### 1.6 References

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; see Chapter 7.

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]...] new-line
```

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 200901.

### 2.3 Accelerator Region Directive

#### Summary

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

#### Syntax

In C, the syntax of the Accelerator region directive is

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

and in Fortran, the syntax is

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

where *clause* is one of the following:

- `if( condition )`
- `copyin( list )`
- `copyout( list )`
- `local( list )`

#### Description

Loops within the structured block will be compiled into accelerator kernels. Data will be copied from the host memory to the accelerator 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 accelerator memory will be deallocated.

**Restrictions**

- Accelerator regions may not be nested.
- A program may not branch into or out of an Accelerator 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.
- A variable may not appear in both the local list and either the copyin or copyout list.

### 2.3.1 if clause

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

When an if clause appears, the compiler will generate two copies of the 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.

### 2.3.2 data clauses

The data clauses are optional. For each variable or array used in the 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 accelerator memory to hold the array;
- Whether the value in host memory needs to be copied to the accelerator 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.

The list argument to each data clause is a comma-separated collection of variable names, array names, or subarray specifications. In C, a subarray is an array name followed by a range specification in brackets, such as

```
arr[2:high]
```

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 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
- 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.

2.3.2.1 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 accelerator 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 clause and does not appear in a copyout clause, then the copyin clause declares that the data need not be copied back from the accelerator memory to the host memory, even if those values were changed on the accelerator.

Restrictions
- A variable, array or subarray may only appear once in any copyin clause for a region.
- Only one subarray for an array may appear in any copyin clause for a region.
- A variable, array or subarray may not appear in any copyin clause for a region if the same variable, array, or any subarray of that array appears in a local clause for the region.
- If an array or subarray appears in a copyin clause, and the same array or a subarray of the same array appears in a copyout clause, the compiler will allocate memory to hold the smallest subarray that encloses both subarrays.
- If an array or subarray appears in a copyin clause, and the same array or a subarray of the same array appears in a copyout clause, the compiler will copy the portion of the array specified in the copyin clause from the host memory to the device, and will copy back that portion of the array that appears in the copyout clause.
- If a subarray appears in a copyin clause, and the same array does not appear in any copyout clause, then the compiler only needs to allocate memory to hold that subarray in the accelerator memory.
- The compiler may pad dimensions of allocated arrays or subarrays to improve memory alignment and program performance.

2.3.2.2 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 accelerator 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 clause and does not appear in a copyin clause, then the copyout clause declares that the data need not be copied to the accelerator memory from the host memory, even if those values are used on the accelerator.

Restrictions
• A variable, array or subarray may only appear once in any copyout clause for a region.
• Only one subarray for an array may appear in any copyout clause for a region.
• A variable, array or subarray may not appear in any copyout clause for a region if the same variable, array, or any subarray of that array appears in a local clause for the region.
• If an array or subarray appears in a copyout clause, and the same array or a subarray of the same array appears in a copyin clause, the compiler will allocate memory to hold the smallest subarray that encloses both subarrays.
• If an array or subarray appears in a copyout clause, and the same array or a subarray of the same array appears in a copyin clause, the compiler will copy the portion of the array specified in the copyin clause from the host memory to the device, and will copy back that portion of the array that appears in the copyout clause.
• If a subarray appears in a copyout clause, and the same array does not appear in any copyin clause, then the compiler only needs to allocate memory for that subarray in the accelerator memory.
• The compiler may pad dimensions of allocated arrays or subarrays to improve memory alignment and program performance.

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

Restrictions
• A variable, array or subarray may only appear once in any local clause for a region.
• Only one subarray for an array may appear in any local clause for a region.
• A variable, array or subarray may not appear in any local clause for a region if the same variable, array, or any subarray of that array appears in a copyin or copyout clause for the region.
• If a subarray appears in a local clause, then the compiler only needs to allocate that subarray in the accelerator memory.
• The compiler may pad dimensions of allocated arrays or subarrays, to improve memory alignment and program performance.

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

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

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

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

where clause is one of the following:

- `host [ (width) ]`
- `parallel [ (width) ]`
- `seq [ (width) ]`
- `vector [ (width) ]`
- `shortloop`
- `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 the 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:

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

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

```fortran
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:

```fortran
!$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:

```fortran
ns = ceil(n/16)
!$acc do host
```
```fortran
    do is = 1, n, ns
    !$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 will strip mine the loop to that many strips, and determine an appropriate schedule for the remaining loop.

### 2.4.1.2 parallel clause
The **parallel** clause tells the compiler to execute this loop in parallel mode on the accelerator. There may be a 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 will strip mine the loop and determine an appropriate schedule for the remaining loop.

### 2.4.1.4 vector clause
The **vector** clause tells the compiler to execute this loop in vector mode on the accelerator. There may be a 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.

### Restrictions
- If two or more loop scheduling clauses appear on a single loop mapping directive, all but one must have a *width* argument.
- 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 shortloop clause
The **shortloop** clause tells the compiler that any target-specific limit on the number of iterations supported in a **parallel** or **vector** loop is satisfied, either because the loop trip count or the value of the *width* expression is small enough. It is an error to use the **shortloop** clause if those limits may be violated.

### 2.4.3 private clause
The **private** clause is used to declare that the variables, arrays or subarrays in the **list** need to be allocated in the accelerator 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 accelerator 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.

**2.4.4 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 region and loop mapping directive is a shortcut for specifying a loop directive nested immediately inside an accelerator `region` directive. The meaning is identical to explicitly specifying a `region` construct containing a loop directive. Any clause that is allowed on a 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 region.
- The restrictions for the region directive and the loop directive apply.
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.

3.2 Runtime Library Routines

3.1.1 accel_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:

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

Fortran:

```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. The
possible values for the argument 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**.

3.1.2 **accel_set_device_num**

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

Format
C:

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

Fortran:

```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.
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_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
```
5. Installation and Supported Platforms

The PGI Accelerator Technology Preview compilers target NVIDIA CUDA-enabled products. Information on particular products can be found at the NVIDIA website:


One of the supported NVIDIA cards will need to be attached to a Linux-based x86-64 machine in order to use the PGI Accelerator Technology.

PGI Fortran or C compilers, version 8.0 or greater, will be required to access the accelerator technology previewed in this document. In addition, a special license is needed to enable that technology with the compilers. Visit http://www.pgroup.com/accelerate if you are interested in using an early preview of the PGI Accelerator Technology.
6. Restrictions & Limitations

6.1 Supported Directives

The PGI Accelerator Technology Preview will support basic directives described in sections 4.3 and 4.4. Subarrays in copyin, copyout, local, and private clauses are not supported. The shortloop clause is ignored. The width expression in all loop scheduling clause is limited to compile-time constants, and the width expression for a vector clause must be a multiple of 16. Function calls within accelerator regions are not currently supported, unless they are inlined.

6.2 Programming Considerations

Arithmetic on many accelerators is not fully IEEE compliant; floating point arithmetic may not support all rounding modes. Floating point arithmetic operations may not be bit-exact for some transcendental functions.

Due to performance reasons double precision arithmetic is not currently supported, even if there is hardware support for double precision.

The compiler may generate code to allocate all the memory needed on the accelerator for all the loops in an accelerator region; if there is not enough memory for all the data, the generated code may fail.

6.3 Hardware Limitations for NVIDIA GPUs

The product of all the width arguments of all nested vector loops must not exceed 512.

The product of all the width arguments of all nested parallel loops must not exceed 65535.

If there is more than one CUDA-capable GPU attached, the default behavior is to use the one with the highest compute capability.
7. Basic Compiler Usage and Examples

7.1 Using the PGI Accelerator Compilers

The basic switch to enable the PGI Accelerator Technology is

- ta=nvidia[,analysis]

which chooses the target NVIDIA GPU.

The analysis option performs the accelerator analysis, but does not produce accelerator device-specific code. Other information:

- Use the –Minfo=accel option to see information regarding accelerator code generation.
- Use –Minfo=intensity to see information about computational intensity for loops.
- The –ta option implies an optimization level of at least –O2.
- With Fortran, use the pgf95 compiler driver, even if the code is written in Fortran 77.
- With C, use the pgcc compiler driver
- With C, the –Mfcon option will compile floating point constants as single precision.
- With C, the –Msafeptr option may help the compiler optimize accelerator-designated regions
- With C99, the restrict pointer keyword may help the compiler optimize accelerator-designated regions

7.2 A Simple Example

Examine the following sample code, named test.f:

```f95
subroutine sum( a, b, c, n)
    real*4 a(10,10), b(10,10), c(10,10)
    integer i,j,n
    !$acc region
    do i = 1, n    ! line 7
        do j = 1,n   ! line 8
            c(i,j) = a(i,j)+ b(i,j)
        enddo
    enddo
    !$acc end region
end

program main
    integer i, j, n
    real*4 input_a(10,10), input_b(10,10), ouput_c(10,10)
    n = 10
    do i=1,n
        do j=1,n
            input_a(i,j) = i
            input_b(i,j) = j
        enddo
    enddo
    call sum(input_a,input_b,output_c,n)
    do i=1,n
        do j=1,n
```

PGI Fortran & C Accelerator Compilers

19
Do we want to use an accelerator to perform the arithmetic computations in the subroutine \texttt{sum}? One measure is computational (or compute) intensity, where a larger number indicates that more arithmetic work is being done compared to memory traffic. The PGI compiler will estimate the compute intensity for loops when the \texttt{–Minfo=intensity} flag is used. For this small example the compute intensity is low (about .33 for both the inner and outer loop), but its simplicity is useful for illustrative purposes. We can first compile the code, without generating GPU code, to see what the compute intensity is:

\begin{verbatim}
pgf95 -O test.f -Minfo=intensity –o accel_test
\end{verbatim}

We see this output:

\begin{verbatim}
sum:
  7, Intensity = (n*n)/((n*n)+((n*n)+(n*n)))
  8, Intensity = 0.33
\end{verbatim}

The symbolic intensity for line 7 is due to unknown loop limits. In this case, no matter what the value of \( n \) is, the expression reduces to .33, the same as the inner loop. These compute intensity numbers are low, and would generally indicate that using an accelerator would not be profitable. However, for demonstration purposes we will next show how to take advantage of the accelerator device.

Now we compile the same routine to generate GPU code as follows:

\begin{verbatim}
pgf95 -O -ta=nvidia test.f -Minfo=accel –o accel_test
\end{verbatim}

and get this resultant output:

\begin{verbatim}
sum:
  8, Kernel schedule is 7(parallel), 8(parallel), 8(vector(16)), 7(vector(16))
\end{verbatim}

Note that a message about the kernel schedule shows that code is produced for the accelerator device. Without such a message (using \texttt{–Minfo=accel}), accelerator code has not been generated (and often information explaining why accelerator code has not been generated will be available). Now the executable \texttt{accel_test} can be run as normal on a Linux x86-64 platform (assuming it has an attached NVIDIA accelerator device).

### 7.3 Matrix Multiplication

A more realistic Fortran example is provided by examining matrix multiply code. In this instance we will have two files, \texttt{mmdriv.f90}, which is the driver program for the matrix multiply routine, and \texttt{mm.f90}, which provides the computational kernel for the application. Here is the code for each one:
program matrix_multiply
  use mymm
  real, allocatable :: a(:,:), b(:,:), c(:,:)
  real :: rmsec, rnn
  integer :: j, i
  print *, 'enter array size'
  read *, nn
  print *, 'enter the number of iterations to run'
  read *, iters
  if( nn .le. 1 ) then
    print *, 'invalid array size: ', nn
    stop
  endif
  if( iters .le. 1 ) then
    iters = 1
  endif
  allocate(a(nn,nn))
  allocate(b(nn,nn))
  allocate(c(nn,nn))

  do j = 1,nn
    do i = 1,nn
      b(i,j) = i*1000.0 + j
      c(i,j) = i+j
      a(i,j) = -1.0
    enddo
  enddo

  do itime = 1,iters
    call cpu_time(r1);
    call mm1( a, b, c, nn )
    call cpu_time(r2)
    rmsec = 1000.0*(r2-r1)
    print *, " cpu_time milliseconds=", rmsec
    print *, " array size =", nn
    rnn = nn
    print *, " MFLOP =", rnn*rnn*rnn*2.0
    print *, " MFLOP/S =",
    rnn*rnn*rnn*2.0 /(rmsec*1000.0)
  enddo
end program
Without an accelerator device one may just compile and run this program on the host as follows:

```
pgf95 -c -fast mm.f90
pgf95 -c -fast mmdriv.f90
pgf95 mm.o mmdriv.o -o mm_out
```

Running this program may result in the following output using array sizes of 4000 (the number of iterations should not matter for a host program, but with an accelerator device the first execution may require loading to the device):

```
enter array size
4000
enter the number of iterations to run
2
cpu_time milliseconds= 69159.43
array size = 4000
MFLOP = 1.2800000E+11
MFLOP/S = 1850.796
```

When trying to identify regions or loops within this computational code that are suitable for offloading to an accelerator we first try compiling with the `-Minfo=intensity` option:

```
pgf90 -c -fast -Minfo=intensity mm.f90
```
How do we interpret these intensity messages? Working from the inside out, line 15 information represents the work done within the \( i \) loop. There are two floating point arithmetic operations (an add and a multiply), and 3 floating point array references that are variant with respect to the loop index \( i \). The array reference \( c(k,j) \) is invariant with respect to the iterations of the inner loop, controlled by the \( i \) variable, and thus does not affect the computational intensity calculation for this inner loop. Therefore we have 2 arithmetic operations and 3 memory accesses, resulting in a ratio of 0.67.

The next outer loop, the \( k \) loop, is more complicated since it contains the inner loop. And now all the array references matter since the \( c(k,j) \) reference now varies with respect to the \( k \) index variable (and note that the \( b(k,i) \) reference varies with respect to both the \( k \) loop and the \( i \) loop). In these more complicated cases the number of times each loop is actually executed—the trip count—is an important factor in the calculation of computational intensity. The symbolic representation for computational intensity at line 14 involves the trip count of two loops, but both loops have a tripcount of \( m \), a parameter passed into the \texttt{mm1} routine. We can see in this example that the value passed in is what the user initially enters when running this program. As an example, let us assume that the user runs this program and inputs 50 for the array size and 2 for the number of iterations. In our calculation above, with \( m=50 \), the computational intensity for line 14 becomes 1.89.

The loop at line 11 has a computational intensity of 0 since there are no arithmetic operations being performed. The outer loop at line 10 contains information about the entire loop nest, and substituting \( m=50 \) into the symbolic expression yields a computational intensity number of 20, certainly large enough to consider offloading the work within this loop to an accelerator.

However, one may often not know the trip counts of important loops. A useful tool is profile feedback, where instrumentation may allow details at execution time to be recorded for future use. In this example profile feedback instrumentation is invoked as follows, using the compiler option \texttt{-Mpfi}:

\begin{verbatim}
pgf95 -c -Mpfi -fast mmdrv.f90
pgf95 -c -Mpfi -fast mm.f90
pgf95 -Mpfi mmdrv.o mm.o -o mm.out
\end{verbatim}

These compilation lines tell the compiler to add instrumentation to the code so that when the program is executed a record is kept on which sections of the code are actually executed, and how often. It is important to note that code performance may noticeably degrade since instrumentation is added to the binary; thus the use of profile feedback should be restricted to cases where the profile information is the goal of execution.

Running the \texttt{mm.out} binary creates a file named \texttt{pgfi.out}; this can be used when compiling the code again, using the profile feedback optimization flag \texttt{-Mpfo}:

\begin{verbatim}
pgf90 -c -Mpfo -fast -Minfo=intensity mm.f90
\end{verbatim}
These results confirm the earlier calculations done by hand, but more importantly show that the matrix multiply kernel is a good candidate for offloading to an accelerator (and this conclusion is when we input the array size as 50; it will be considerably higher with an array size of 4000). As we saw in the Section 7.2 example, directives are placed around the computational kernel, in this example around the triply-nested loop within subroutine mm1. Note that the presence of the directives does not affect the compilation of the program in any way until we instruct the compiler with the `–ta=nvidia` option.

```
pgf95 -c -fast -ta=nvidia -Minfo=accel mm.f90
```

As before, we use the `–Minfo=accel` option to receive information on accelerator scheduling. Here we see that kernels are generated for the accelerator device. We also need to compile the driver module, then link the program together:

```
pgf95 -c -fast mmdriv.f90
pgf95 -ta=nvidia mm.o mmdriv.o -o mm_out
```

Note that the `–ta=nvidia` switch is not required when compiling the driver routine, since no accelerator pragmas appear in that file. However, the flag is required on the link line. This insures that the accelerator runtime libraries are linked in.

Now, with the computational kernel running on the NVIDIA accelerator device, we get this result when mm_out is run with array size of 4000:

```
4000
enter the number of iterations to run
3

cpu_time milliseconds= 1987.101
  array size = 4000
  MFLOP = 1.2800000E+11
  MFLOP/S = 64415.45

cpu_time milliseconds= 1929.738
  array size = 4000
  MFLOP = 1.2800000E+11
  MFLOP/S = 66330.25

cpu_time milliseconds= 1927.009
  array size = 4000
  MFLOP = 1.2800000E+11
  MFLOP/S = 66424.17
```
With this simple example we see a speedup of over 30 using the accelerator versus just running on the host. The first iteration is a bit slower since the accelerator needs to be initialized.