# PGI® COMPILERS &TOOLS

## OPENACC GETTING STARTED GUIDE

Version 2019

**nvidia**

# TABLE OF CONTENTS

# LIST OF TABLES

# Chapter 1.
# OVERVIEW

The OpenACC Application Program Interface is a collection of compiler directives and runtime routines that allow you, the programmer, to specify loops and regions of code in standard C, C++ and Fortran that you want offloaded from a host CPU to an attached accelerator, such as a GPU. The OpenACC API was designed and is maintained by an industry consortium. See the OpenACC website, http://www.openacc.org for more information about the OpenACC API.

This Getting Started guide helps you prepare your system for using the PGI OpenACC implementation, and provides examples of how to write, build and run programs using the OpenACC directives. See the PGI OpenACC webpage, pgicompilers.com/openacc for more information about the PGI OpenACC implementation.

## 1.1. Terms and Definitions

Throughout this document certain terms have very specific meaning:

▸ OpenACC is a parallel programming standard describing a set of compiler directives which can be applied to standard C, C++, and Fortran to specify regions of code for offloading from a host CPU to an attached accelerator.

▸ A directive is, 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. This document uses the term directives for either Fortran directive comments or C/C++ pragmas. Features specific to "Fortran directives" and "C pragmas" are called out as such.

▸ PGCC, PGC++, and PGFORTRAN are the names of the PGI compiler products.

▸ pgcc, pgc++ and pgfortran are the names of the PGI compiler drivers. pgfortran may also be spelled pgf90 and pgf95. The PGI C++ compiler, named pgc++, uses GNU-compatible naming conventions.

▸ CUDA stands 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.

▸ LLVM is a compiler infrastructure. Under certain circumstances, PGI compilers may produce an intermediate representation of programs for use by LLVM compiler back-ends.

## 1.2. System Prerequisites

Using this release of PGI OpenACC API implementation requires the following:

▸ An x86-64/x64 system running Linux or Microsoft Windows or an OpenPOWER system running Linux. See the PGI Compatibility and Installation webpage at https://www.pgroup.com/products/index.htm?tab=compat for information about the PGI-supported releases.
▸ For targeting GPUs:

  ▸ **NVIDIA:** A CUDA-enabled NVIDIA GPU and an installed driver. For NVIDIA CUDA, the driver should be version 8.0 or later. (See the NVIDIA CUDA webpage at http://www.nvidia.com/cuda for more information).

## 1.3. Prepare Your System

To enable OpenACC, follow these steps:

1. Download the latest 19.10 packages from the PGI download webpage at pgicompilers.com/downloads.

2. Install the downloaded package.

3. Add the installed bin directory on your path.

4. Run pgaccelinfo to see that your GPU and drivers are properly installed and available. For NVIDIA, you should see output that looks something like the following:

```
CUDA Driver Version:          9020
NVRM version:                 NVIDIA UNIX x86_64 Kernel Module  396.26  Mon
 Apr 30 18:01:39 PDT 2018

Device Number:                0
Device Name:                  Tesla V100-PCIE-16GB
Device Revision Number:       7.0
Global Memory Size:           16945512448
Number of Multiprocessors:    80
Concurrent Copy and Execution: Yes
Total Constant Memory:        65536
Total Shared Memory per Block: 49152
Registers per Block:          65536
Warp Size:                    32
Maximum Threads per Block:    1024
Maximum Block Dimensions:     1024, 1024, 64
Maximum Grid Dimensions:      2147483647 x 65535 x 65535
Maximum Memory Pitch:         2147483647B
Texture Alignment:            512B
Clock Rate:                   1380 MHz
Execution Timeout:            No
Integrated Device:            No
```

```
Can Map Host Memory:          Yes
Compute Mode:                 default
Concurrent Kernels:           Yes
ECC Enabled:                  Yes
Memory Clock Rate:            877 MHz
Memory Bus Width:             4096 bits
L2 Cache Size:                6291456 bytes
Max Threads Per SMP:          2048
Async Engines:                7
Unified Addressing:           Yes
Managed Memory:               Yes
Concurrent Managed Memory:    Yes
Preemption Supported:         Yes
Cooperative Launch:           Yes
  Multi-Device:               Yes
PGI Default Target:           -ta=tesla:cc70
```

This tells you the driver version, the name of the GPU (or GPUs, if you have more than one), the available memory, the **-ta** command line flag to target this GPU, and so on.

# 1.4. Supporting Documentation and Examples

You may want to consult the latest OpenACC 2.5 specification available at the OpenACC website, http://www.openacc.org. Simple examples appear in Using OpenACC with the PGI Compilers.

# Chapter 2.
# USING OPENACC WITH THE PGI COMPILERS

The OpenACC directives are enabled by adding the `-acc` or the `-ta=[target]` flag to the PGI compiler command line. This release targets OpenACC to NVIDIA GPUs. `[-ta=tesla]`

Refer to Implemented Features for a discussion about using OpenACC directives or the `-acc` flag with object files compiled with previous PGI releases using the PGI Accelerator directives.

This release includes support for almost all of the OpenACC 2.5 specification. Refer to Implemented Features for details about which features are supported in this release, and what features are coming in updates over the next few months.

## 2.1. OpenACC Directive Summary

PGI now supports the following OpenACC directives:

**Parallel Construct**

Defines the region of the program that should be compiled for parallel execution on the accelerator device.

**Kernels Construct**

Defines the region of the program that should be compiled into a sequence of kernels for execution on the accelerator device.

**Data 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.

### Enter Data and Exit Data Directives

The Enter Data directive defines data, typically arrays, that should be allocated in the device memory for the duration of the program or until an exit data directive that deallocates the data, and whether data should be copied from the host to the device memory at the enter data directive.

The Exit Data directive defines data, typically arrays, that should be deallocated in the device memory, and whether data should be copied from the device to the host memory.

### Host_Data Construct

Makes the address of device data available on the host.

### Loop Directive

Describes what type of parallelism to use to execute the loop and declare loop-private variables and arrays and reduction operations. Applies to a loop which must appear on the following line.

### Combined Parallel and Loop Directive

Is a shortcut for specifying a loop directive nested immediately inside an accelerator parallel directive. The meaning is identical to explicitly specifying a parallel construct containing a loop directive.

### Combined Kernels and Loop Directive

Is a shortcut for specifying a loop directive nested immediately inside an accelerator kernels directive. The meaning is identical to explicitly specifying a kernels construct containing a loop directive.

### Cache Directive

Specifies array elements or subarrays that should be fetched into the highest level of the cache for the body of a loop. Must appear at the top of (inside of) the loop.

### Declare Directive

Specifies 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.

Specifies 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.

Creates a visible device copy of the variable or array.

**Update Directive**

Used during the lifetime of accelerator data 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.

**Routine Directive**

Used to tell the compiler to compile a given procedure for an accelerator as well as the host. In a file or routine with a procedure call, the routine directive tells the implementation the attributes of the procedure when called on the accelerator.

As of PGI 14.9, an `acc routine` directive with no parallelism clause (`gang`, `worker` or `vector`) will be treated as if the `seq` clause were present.

**Wait Directive**

Specifies to wait until all operations on a specific device async queue or all async queues are complete.

For more information on each of these directives and which clauses they accept, refer to the 'Using an Accelerator' section in the respective PGI User's Guide for your product.

## 2.2. CUDA Toolkit Versions

The PGI compilers use NVIDIA's CUDA Toolkit when building programs for execution on an NVIDIA GPU. Every PGI installation package puts the required CUDA Toolkit components into a PGI installation directory called `2019/cuda`.

An NVIDIA CUDA driver must be installed on a system with a GPU before you can run a program compiled for the GPU on that system. PGI products do not contain CUDA Drivers. You must download and install the appropriate CUDA Driver from NVIDIA. The CUDA Driver version must be at least as new as the version of the CUDA Toolkit with which you compiled your code.

The PGI tool `pgaccelinfo` prints the driver version as its first line of output. You can use it to find out which version of the CUDA Driver is installed on your system.

PGI 19.10 includes the following versions of the CUDA Toolkit:

▸   CUDA 9.2
▸   CUDA 10.0
▸   CUDA 10.1

You can let the compiler pick which version of the CUDA Toolkit to use or you can instruct it to use a particular version. The rest of this section describes all of your options.

If you do not specify a version of the CUDA Toolkit, the compiler uses the version of the CUDA Driver installed on the system on which you are compiling to determine which CUDA Toolkit to use. This auto-detect feature was introduced in the PGI 18.7

release; auto-detect is especially convenient when you are compiling and running your application on the same system. In the absence of any other information, the compiler will look for a CUDA Toolkit version in the PGI `2019/cuda` directory that matches the version of the CUDA Driver installed on the system. If a match is not found, the compiler searches for the newest CUDA Toolkit version that is not newer than the CUDA Driver version. If there is no CUDA Driver installed, the PGI 19.10 compilers fall back to the default of CUDA 9.2.

If the only PGI compiler you have installed is PGI 19.10, then:

▸ If your CUDA Driver is 10.1, the compilers use CUDA Toolkit 10.1.
▸ If your CUDA Driver is 10.0, the compilers use CUDA Toolkit 10.0.
▸ If your CUDA Driver is 9.2, the compilers use CUDA Toolkit 9.2.
▸ If your CUDA Driver is 9.1, the compilers will issue an error that CUDA Toolkit 9.1 was not found; CUDA Toolkit 9.1 is not bundled with PGI 19.10
▸ If you do not have a CUDA driver installed on the compilation system, the compilers use the default CUDA Toolkit version 9.2.
▸ If your CUDA Driver is newer than CUDA 10.1, the compilers will still use the CUDA Toolkit 10.1. The compiler selects the newest CUDA Toolkit it finds that is not newer than the CUDA Driver.

You can change the compiler's default selection for CUDA Toolkit version using one of the following methods:

▸ Use a compiler option. Add the `cudaX.Y` sub-option to `-Mcuda` or `-ta=tesla` where `X.Y` denotes the CUDA version. For example, to compile a C file with the CUDA 9.2 Toolkit you would use:
```
pgcc -ta=tesla:cuda9.2
```

Using a compiler option changes the CUDA Toolkit version for one invocation of the compiler.

▸ Use an rcfile variable. Add a line defining `DEFCUDAVERSION` to the `siterc` file in the installation `bin/` directory or to a file named `.mypgirc` in your home directory. For example, to specify the CUDA 9.2 Toolkit as the default, add the following line to one of these files:
```
set DEFCUDAVERSION=9.2;
```

Using an rcfile variable changes the CUDA Toolkit version for all invocations of the compilers reading the rcfile.

When you specify a CUDA Toolkit version, you can additionally instruct the compiler to use a CUDA Toolkit installation different from the defaults bundled with the current PGI compilers. While most users do not need to use any other CUDA Toolkit installation than those provided with PGI, situations do arise where this capability is needed. Developers working with pre-release CUDA software may occasionally need to test with a CUDA Toolkit version not included in a PGI release. Conversely, some developers might find a need to compile with a CUDA Toolkit older than the oldest CUDA Toolkit installed with a PGI release. For these users, PGI compilers can interoperate with components from a CUDA Toolkit installed outside of the PGI installation directories.

PGI tests extensively using the co-installed versions of the CUDA Toolkits and fully supports their use. Use of CUDA Toolkit components not included with a PGI install is done with your understanding that functionality differences may exist.

The ability to compile with a CUDA Toolkit other than the versions installed with the PGI compilers is supported on all platforms; on the Windows platform, this feature is supported for CUDA Toolkit versions 9.2 and newer.

To use a CUDA toolkit that is not installed with a PGI release, such as CUDA 9.1 with PGI 19.10, there are three options:

▸ Use the rcfile variable `DEFAULT_CUDA_HOME` to override the base default
  ```
  set DEFAULT_CUDA_HOME = /opt/cuda-9.1;
  ```
▸ Set the environment variable `CUDA_HOME`
  ```
  export CUDA_HOME=/opt/cuda-9.1
  ```
▸ Use the compiler compilation line assignment CUDA_HOME=
  ```
  pgfortran CUDA_HOME=/opt/cuda-9.1
  ```

The PGI compilers use the following order of precedence when determining which version of the CUDA Toolkit to use.

1. If you do not tell the compiler which CUDA Toolkit version to use, the compiler picks the CUDA Toolkit from the PGI installation directory `2019/cuda` that matches the version of the CUDA Driver installed on your system. If the PGI installation directory does not contain a direct match, the newest version in that directory which is not newer than the CUDA driver version is used. If there is no CUDA driver installed on your system, the compiler falls back on an internal default; in PGI 19.10, this default is CUDA 9.2.
2. The rcfile variable `DEFAULT_CUDA_HOME` will override the base default.
3. The environment variable `CUDA_HOME` will override all of the above defaults.
4. The environment variable `PGI_CUDA_HOME` overrides all of the above; it is available for advanced users in case they need to override an already-defined `CUDA_HOME`.
5. A user-specified `cudaX.Y` sub-option to `-Mcuda` and `-ta=tesla` will override all of the above defaults and the CUDA Toolkit located in the PGI installation directory `2019/cuda` will be used.
6. The compiler compilation line assignment `CUDA_HOME=` will override all of the above defaults (including the `cudaX.Y` sub-option).

## 2.3. Compute Capability

The compilers can generate code for NVIDIA GPU compute capabilities 3.0 through 7.5. The compilers construct a default list of compute capabilities that matches the compute capabilities supported by the GPUs found on the system used in compilation. If there are no GPUs detected, the compilers select cc35, cc50, cc60, and cc70.

You can override the default by specifying one or more compute capabilities using either command-line options or an `rcfile`.

To change the default with a command-line option, provide a comma-separated list of compute capabilities to `-ta=tesla:` for OpenACC or `-Mcuda=` for CUDA Fortran.

To change the default with an `rcfile`, set the **DEFCOMPUTECAP** value to a blank-separated list of compute capabilities in the siterc file located in your installation's bin directory:

```
set DEFCOMPUTECAP=60 70;
```

Alternatively, if you don't have permissions to change the `siterc` file, you can add the **DEFCOMPUTECAP** definition to a separate `.mypgirc` file (`mypgi_rc` on Windows) in your home directory.

The generation of device code can be time consuming, so you may notice an increase in compile time as the number of compute capabilities increases.

# 2.4. C Structs in OpenACC

Static arrays of struct and pointers to dynamic arrays of struct have long been supported with the PGI Accelerator compilers.

```
typedef struct{
    float x, y, z;
}point;

extern point base[1000];

void vecaddgpu( point *restrict r, int n ){
    #pragma acc parallel loop present(base) copyout(r[0:n])
    for( int i = 0; i < n; ++i ){
 r[i].x = base[i].x;
 r[i].y = sqrtf( base[i].y*base[i].y + base[i].z*base[i].z );
 r[i].z = 0;
    }
}
```

A pointer to a scalar struct is treated as a one-element array, and should be shaped as **r[0:1]**.

PGI 14.4 and later releases include support for static arrays and pointers to dynamic arrays within a struct. In either case, the entire struct must be placed in device memory, by putting the struct itself in an appropriate data clause.

```
typedef struct{
    base[1000];
    int n;
    float *x, *y, *z;
}point;

extern point A;

void vecaddgpu(){
    #pragma acc parallel loop copyin(A) \
 copyout(A.x[0:A.n], A.y[0:A.n], A.z[0:A.n])
    for( int i = 0; i < A.n; ++i ){
 A.x[i] = A.base[i];
 A.y[i] = sqrtf( A.base[i] );
 A.z[i] = 0;
    }
}
```

In this example, the struct **A** is copied to the device, which copies the static array member **A.base** and the scalar **A.n**. The dynamic members **A.x**, **A.y** and **A.z** are then copied to the device. The struct **A** should be copied before its dynamic members, either

by placing the struct in an earlier data clause, or by copying or creating it on the device in an enclosing data region or dynamic data lifetime. If the struct is not present on the device when the dynamic members are copied, the accesses to the dynamic members, such as `A.x[i]`, on the device will be invalid, because the pointer `A.x` will not get updated.

A pointer to a struct is treated as a single element array. If the struct also contains pointer members, you should copy the struct to the device, then create or copy the pointer members:

```
typedef struct{
    int n;
    float *x, *y, *z;
 }point;

void vecaddgpu( point *A, float* base ){
    #pragma acc parallel loop copyin(A[0:1]) \
 copyout(A->x[0:A->n], A->y[0:A->n], A->z[0:A->n]) \
 present(base[0:A->n])
    for( int i = 0; i < A->n; ++i ){
 A->x[i] = base[i];
 A->y[i] = sqrtf( base[i] );
 A->z[i] = 0;
    }
}
```

Be careful when copying structs containing pointers back to the host. On the device, the pointer members will get updated with device pointers. If these pointers get copied back to the host struct, the pointers will be invalid on the host.

When creating or copying a struct on the device, the whole struct is allocated. There is no support for allocating a subset of a struct, or only allocating space for a single member.

Structs and pointer members can be managed using dynamic data directives as well:

```
typedef struct{
    int n;
    float *x, *y, *z;
 }point;

void move_to_device( point *A ){
    #pragma acc enter data copyin(A[0:1])
    #pragma acc enter data create(A->x[0:A->n], A->y[0:A->n], A->z[0:A->n])
}

void move_from_device( point* A ){
    #pragma acc enter data copyout(A->x[0:A->n], A->y[0:A->n], A->z[0:A->n])
    #pragma acc enter data delete(A[0:1])
}

void vecaddgpu( point *A, float* base ){
    #pragma acc parallel loop present(A[0:1]) \
 present(A->x[0:A->n], A->y[0:A->n], A->z[0:A->n]) \
 present(base[0:A->n])
    for( int i = 0; i < A->n; ++i ){
 A->x[i] = base[i];
 A->y[i] = sqrtf( base[i] );
 A->z[i] = 0;
    }
}
```

## 2.5. C++ Classes in OpenACC

PGI 14.4 and later releases include support for C++ classes, including static array class members, member pointers to dynamic arrays, and member functions and operators. Usually, the class itself must be copied to device memory as well, by putting the class variable in a data clause outside the class, or the appropriately shaped **this[0:1]** reference in a data clause within the class. The entire class will be allocated in device memory.

```cpp
// my managed vector datatype
template<typename elemtype> class myvector{
    elemtype* data;
    size_t size;
public:
    myvector( size_t size_ ){ // constructor
        size = size_;
        data = new elemtype[size];
    }
    todev(){ // move to device
        #pragma acc enter data copyin(this[0:1], data[0:size])
    }
    fromdev(){ // remove from device
        #pragma acc exit data delete( data[0:size], this[0:1])
    }
    void updatehost(){ // update host copy of data
        #pragma acc update self( data[0:size] )
    }
    void updatedev(){ // update device copy of data
        #pragma acc update device( data[0:size] )
    }
    ~myvector(){ // destructor from host
        delete[] data;
    }
    inline elemtype & operator[] (int i) const { return data[i]; }
    // other member functions
};
```

In this class, **this** is copied to the device before **data**, so the pointer to **data** on the device will get updated. This is called an "attach" operation; the **class myvector** pointer **data** is attached to the device copy of the **data** vector.

Another class always creates device data along with host data:

```cpp
// my managed host+device vector datatype
template<typename elemtype> class hdvector{
    elemtype* data;
    size_t size;
public:
    hdvector( size_t size_ ){ // constructor
        size = size_;
        data = new elemtype[size];
        #pragma acc enter data copyin(this[0:1]) create(data[0:size])
    }
    void updatehost(){ // update host copy of data
        #pragma acc update self( data[0:size] )
    }
    void updatedev(){ // update device copy of data
        #pragma acc update device( data[0:size] )
    }
    ~hdvector(){ // destructor from host
        #pragma acc exit data delete( data[0:size], this[0:1] )
        delete[] data;
```

```
    }
    inline elemtype & operator[] (int i) const { return data[i]; }
    // other member functions
};
```

The constructor copies the class in, so the **size** value will get copied, and creates (allocates) the **data** vector.

A slightly more complex class includes a copy constructor that makes a copy of the data pointer instead of a copy of the data:

```
#include <openacc.h>
// my managed vector datatype
template<typename elemtype> class dupvector{
    elemtype* data;
    size_t size;
    bool iscopy;
public:
    dupvector( size_t size_ ){ // constructor
        size = size_;
        data = new elemtype[size];
        iscopy = false;
        #pragma acc enter data copyin(this[0:1]) create(data[0:size])
    }
    dupvector( const dupvector &copyof ){ // copy constructor
        size = copyof.size;
        data = copyof.data;
        iscopy = true;
        #pragma acc enter data copyin(this[0:1])
        acc_attach( (void**)&data );
    }
    void updatehost(){ // update host copy of data
        #pragma acc update self( data[0:size] )
    }
    void updatedev(){ // update device copy of data
        #pragma acc update device( data[0:size] )
    }
    ~dupvector(){ // destructor from host
        if( !iscopy ){
            #pragma acc exit data delete( data[0:size] )
            delete[] data;
        }
        #pragma acc exit data delete( this[0:1] )
    }
    inline elemtype & operator[] (int i) const { return data[i]; }
    // other member functions
};
```

We added a call to the PGI OpenACC runtime routine, **acc_attach**, in the copy constructor. This routine is a PGI addition to the OpenACC API; it takes the address of a pointer, translates the address of that pointer as well as the contents of the pointer, and stores the translated contents into the translated address on the device. In this case, it attaches the data pointer copied from the original class on the device to the copy of this class on the device.

In code outside the class, data can be referenced in compute clauses as normal:

```
dupvector<float> v = new dupvector<float>(n);
dupvector<float> x = new dupvector<float>(n);
...
#pragma acc parallel loop present(v,x)
 for( int i = 0; i < n; ++i ) v[i] += x[i];
```

This example shows references to the **v** and **x** classes in the parallel loop construct. The **operator[]** will normally be inlined. If it is not inlined or inlining is disabled, the compiler will note that the operator is invoked from within an OpenACC compute region and compile a device version of that operator. This is effectively the same as implying a **#pragma acc routine seq** above the operator. The same is true for any function in C++, be it a class member function or standalone function: if the function is called from within a compute region, or called from a function which is called within a compute region, and there is no **#pragma acc routine**, the compiler will treat it as if it was prefixed by **#pragma acc routine seq**. When you compile the file and enable **–Minfo=accel**, you will see this with the message:

```
T1 &dupvector<T1>::operator [](int) const [with T1=float]:
        35, Generating implicit acc routine seq
```

In the above example, the loop upper bound is the simple variable **n**, not the more natural class member **v.size**. In this PGI release, the loop upper bound for a parallel loop or kernels loop must be a simple variable, not a class member. This limitation will be fixed in a future release.

The class variables appear in a **present** clause for the parallel construct. The normal default for a compute construct would be for the compiler to treat the reference to the class as **present_or_copy**. However, if the class instance were not present, copying just the class itself would not copy the dynamic data members, so would not provide the necessary behavior. Therefore, when refering to class objects in a compute construct, you should put the class in a **present** clause.

Class member functions may be explicitly invoked in a parallel loop:

```
template<typename elemtype> class dupvector{
    ...
    void inc1( int i, elemtype y ){
        data[i] += y;
    }
}
...
#pragma acc parallel loop present(v,x)
    for( int i = 0; i < n; ++i ) v.inc1( i, x[i] );
```

As discussed above, the compiler will normally inline **inc1**, when optimization is enabled, but will also compile a device version of the function since it is invoked from within a compute region.

A compute construct may contain compute constructs itself:

```
template<typename elemtype> class dupvector{
    ...
    void inc2( dupvector<elemtype> &y ){
        int n = size;
        #pragma acc parallel loop gang vector present(this,y)
        for( int i = 0; i < n; ++i ) data[i] += y[i];
    }
}
...
    v.inc2( x );
```

Note again the loop upper bound of **n**, and the **this** and **y** classes in the **present** clause. A third example puts the parallel construct around the routine, but the loop

itself within the routine. Doing this properly requires you to put an appropriate **acc routine** before the routine definition to call the routine at the right level of parallelism.

```
template<typename elemtype> class dupvector{
    ...
    #pragma acc routine gang
    void inc3( dupvector<elemtype> &y ){
        int n = size;
        #pragma acc loop gang vector
        for( int i = 0; i < n; ++i ) data[i] += y[i];
    }
}
...
    #pragma acc parallel
        v.inc3( x );
```

When the **inc3** is invoked from host code, it will run on the host incrementing host values. When invoked from within an OpenACC parallel construct, it will increment device values.

# 2.6. Fortran Derived Types in OpenACC

Static and allocatable arrays of derived type have long been supported with the PGI Accelerator compilers.

```
module mpoint
type point
    real :: x, y, z
end type
type(point) :: base(1000)
end module

subroutine vecaddgpu( r, n )
 use mpoint
 type(point) :: r(:)
 integer :: n
 !$acc parallel loop present(base) copyout(r(:))
 do i = 1, n
  r(i)%x = base(i)%x
  r(i)%y = sqrt( base(i)%y*base(i)%y + base(i)%z*base(i)%z )
  r(i)%z = 0
 enddo
end subroutine
```

PGI 14.4 and later releases include support for array members of derived types, including static arrays and allocatable arrays within a derived type. In either case, the entire derived type must be placed in device memory, by putting the derived type itself in an appropriate data clause. For this release, the derived type variable itself must appear in a data clause, at least a **present** clause, for any compute construct that directly uses the derived type variable.

```
module mpoint
type point
    real :: base(1000)
    integer :: n
    real, allocatable, dimension(:) :: x, y, z
end type

type(point) :: A
end module
```

```
subroutine vecaddgpu()
 integer :: i
 !$acc parallel loop copyin(A) copyout(A%x,A%y,A%z)
 do i = 1, n
  A%x(i) = A%base(i)
  A%y(i) = sqrt( A%base(i) )
  A%z(i) = 0
 enddo
end subroutine
```

In this example, the derived type **A** is copied to the device, which copies the static array member **A%base** and the scalar **A%n**. The allocatable array members **A%x**, **A%y** and **A%z** are then copied to the device. The derived type variable **A** should be copied before its allocatable array members, either by placing the derived type in an earlier data clause, or by copying or creating it on the device in an enclosing data region or dynamic data lifetime. If the derived type is not present on the device when the allocatable array members are copied, the accesses to the allocatable members, such as **A%x(i)**, on the device will be invalid, because the hidden pointer and descriptor values in the derived type variable will not get updated.

Be careful when copying derived types containing allocatable members back to the host. On the device, the allocatable members will get updated to point to device memory. If the whole derived type gets copied back to the host, the allocatable members will be invalid on the host.

When creating or copying a derived type on the device, the whole derived type is allocated. There is no support for allocating a subset of a derived type, or only allocating space for a single member.

Derived types and allocatable members can be managed using dynamic data directives as well:

```
module mpoint
 type point
    integer :: n
    real, dimension(:), allocatable :: x, y, z
 end type
contains
 subroutine move_to_device( A )
  type(point) :: A
  !$acc enter data copyin(A)
  !$acc enter data create(A%x, A%y, A%z)
 end subroutine

 subroutine move_off_device( A )
  type(point) :: A
  !$acc exit data copyout(A%x, A%y, A%z)
  !$acc exit data delete(A)
 end subroutine
end module

subroutine vecaddgpu( A, base )
 use mpoint
 type(point) :: A
 real :: base(:)
 integer :: i
 !$acc parallel loop present(A,base)
 do i = 1, n
  A%x(i) = base(i)
  A%y(i) = sqrt( base(i) )
  A%z(i) = 0
 enddo
```

```
end subroutine
```

Arrays of derived type, where the derived type contains allocatable members, have not been tested and should not be considered supported for this release. That important feature will be included in an upcoming release.

# 2.7. Fortran I/O

Starting in PGI 15.1, improvements have been made in the support of list-directed PRINT or WRITE statements to the default output unit (PRINT * or WRITE(*,*)). Before PGI 15.1, the output for PRINT or WRITE statements would likely be interleaved between different threads for each item on the PRINT or WRITE statement. That is, if a device kernel contains a PRINT statement, such as this one:

```
print *, 'index = ', i, j
```

the user would observe the character string 'index = ', one after the other from all threads, then the second item, the value of i, then the third item, j, and finally the end-of-line.

Unlike the usual C printf implementation, which prints out a whole line for each thread, there was no indication of which thread prints out which item, and in which order.

Now, starting with PGI 15.1, the PGI Fortran accelerator runtime, shared between CUDA Fortran and OpenACC for NVIDIA targets, buffers up the output and prints an entire line's worth in one operation. In PGI 15.1 through 15.4, the integer and character types were supported. Beginning in PGI 15.5, logical, real and complex types are also supported.

The underlying CUDA printf implementation limits the number of print statements in a kernel launch to 4096. Users should take this limit into account when making use of this feature.

Starting in PGI 15.5, print and write statements in device code are also supported when used with the LLVM code generator -ta=llvm, or -Mcuda=llvm, and in combination with the -mp compiler option.

## 2.7.1. OpenACC PRINT Example

Here is a short example printing character strings, integer, logical, and real data within an OpenACC compute region:

```
program t
integer(4) a(10000)
a = [ (1+i,i=1,10000) ]
!$acc kernels
do i = 1, 10000
  if (a(i)/3000*3000.eq.a(i)) print *," located ",i,a(i),i.gt.5000,a(i)/5.0
end do
!$acc end kernels
end
```

## 2.8. OpenACC Atomic Support

PGI 14.7 and later releases provides full support for atomics in accordance with the 2.0 spec. For example:

```
double *a, *b, *c;
. . .
#pragma acc loop vector
    for (int k = 0; k < n; ++k)
    {
        #pragma acc atomic
        c[i*n+j] += a[i*n+k]*b[k*n+j];
    }
```

PGI 14.4 and later releases include support for CUDA-style atomic operations. The CUDA atomic names can be used in accelerator regions from Fortran, C, and C++. For example:

```
. . .
#pragma acc loop gang
    for (j = 0; j < n1 * n2; j += n2) {
        k = 0;
        #pragma acc loop vector reduction(+:k)
            for (i = 0; i < n2; i++)
                k = k + a[j + i];
            atomicAdd(x, k);
    }
```

## 2.9. OpenACC Declare Data Directive for Global and Fortran Module Variables

PGI 14.7 and later releases support the OpenACC **declare** directive with the **copyin**, **create** and **device_resident** clauses for C global variables and Fortran module variables, for Tesla-target GPUs. This is primarily for use with the OpenACC **routine** directive and separate compilation. The data in the **declare** clauses are statically allocated on the device when the program attaches to the device. Data in a **copyin** clause will be initialized from the host data at that time. A program attaches to the device when it reaches its first data or compute construct, or when it calls the OpenACC **acc_init** routine.

In C, the example below uses a global struct and a global array pointer:

```
struct{
    float a, b;
}coef;
float* x;
#pragma acc declare create(coef,x)
. . .
#pragma acc routine seq
void modxi( int i ){
    x[i] *= coef.a;
}
. . .
void initcoef( float a, float b ){
```

```
        coef.a = a;
        coef.b = b;
        #pragma acc update device(coef)
}
. . .
void allocx( int n ){
        x = (float*)malloc( sizeof(float)*n );
        #pragma acc enter data create(x[0:n])
}
. . .
void modx( int s, int e ){
        #pragma acc parallel loop
        for( int i = s; i < e; ++i ) modxi(i);
}
```

The **declare create(coef,x)** will statically allocate a copy of the struct **coef** and the pointer **x** on the device. In **initcoef** routine, the coefficients are assigned on the host, and the **update** directive copies those values to the device. The **allocx** routine allocates space for the **x** vector on the host, then uses an unstructured data directive to allocate that space on the device as well; because the **x** pointer is already statically present on the device, the device copy of **x** will be updated with the pointer to the device data as well. Finally, the parallel loop calls the routine **modxi**, which refers to the global **x** pointer and **coef** struct. When called on the host, this routine will access the global **x** and **coef** on the host, and when called on the device, such as in this parallel loop, this routine will access the global **x** pointer and **coef** struct on the device.

If the **modxi** routine were in a separate file, the declarations of **coef** and **x** would have the **extern** attribute, but otherwise the code would be the same, as shown below. Note that the **acc declare create** directive is still required in this file even though the variables are declared **extern**, to tell the compiler that these variables are available as externals on the device.

```
extern struct{
        float a, b;
}coef;
extern float* x;
#pragma acc declare create(coef,x)
. . .
#pragma acc routine seq
void modxi( int i ){
        x[i] *= coef.a;
}
```

Because the global variable is present in device memory, it is also in the OpenACC runtime *present* table, which keeps track of the correspondence between host and device objects. This means that a pointer to the global variable can be passed as an argument to a routine in another file, which uses that pointer in a **present** clause. In the following example, the calling routine uses a small, statically-sized global coefficient array:

```
float xcoef[11] = { 1.0, 2.0, 1.5, 3.5, ... 9.0 };
#pragma acc declare copyin(xcoef)
. . .
extern void test( float*, float*, float*, n );
. . .
void caller( float* x, float* y, int n ){
        #pragma acc data copy( x[0:n], y[0:n] )
        {
                . . .
                test( x, y, xcoef, n );
                . . .
```

```
     }
}
```

The **declare copyin** directive tells the compiler to generate code to initialize the device array from the host array when the program attaches to the device. In another file, the procedure **test** is defined, and all of its array arguments will be already present on the device; **x** and **y** because of the data construct, and **xcoef** because it is statically present on the device.

```
void test( float* xx, float* yy, float* cc, int n ){
    #pragma acc data present( xx[0:n], y[00:n], cc[0:11] )
    {
        . . .
        #pragma acc parallel loop
        for( int i = 5; i < n-5; ++i ){
            float t = 0.0;
            for( int j = -5; j <= 5; ++j ){
                t += cc[j+5]*yy[i+j];
            }
            xx[i] /= t;
        }
        . . .
    }
}
```

In Fortran, module fixed-size variables and arrays, and module allocatable arrays which appear in **declare** directives at module scope will be available globally on the CPU as well as in device code. Module allocatable arrays that appear in a **declare create**, **declare copyin** or **declare device_resident** will be allocated in host memory as well as in device memory when they appear in an allocate statement. The compiler manages the actual pointer to the data and a descriptor that contains array lower and upper bounds for each dimension, and the device copy of the pointer will be set to point to the array in device memory.

The following example module contains one fixed size array and an allocatable array, both appearing in a **declare create** clause. The static array **xstat** will be available at any time inside accelerator compute regions or routines.

```
module staticmod
 integer, parameter :: maxl = 100000
 real, dimension(maxl) :: xstat
 real, dimension(:), allocatable :: yalloc
 !$acc declare create(xstat,yalloc)
end module
```

This module may be used in another file that allocates the **yalloc** array. When the allocatable array **yalloc** is allocated, it will be allocated both in host and device memory, and will then be available at any time in accelerator compute regions or routines.

```
subroutine allocit(n)
 use staticmod
 integer :: n
 allocate( yalloc(n) )
end subroutine
```

In another module, these arrays may be used in a compute region or in an accelerator routine:

```
module useit
 use staticmod
contains
 subroutine computer( n )
  integer :: n
  integer :: i
  !$acc parallel loop
   do i = 1, n
    yalloc(i) = iprocess( i )
   enddo
 end subroutine
 real function iprocess( i )
  !$acc routine seq
  integer :: i
  iprocess = yalloc(i) + 2*xstat(i)
 end function
end module
```

# 2.10. OpenACC Error Handling

The OpenACC specification provides a mechanism to allow you to intercept errors triggered during execution on a GPU and execute a specific routine in response before the program exits. For example, if an MPI process fails while allocating memory on the GPU, the application may want to call `MPI_Abort` to shut down all the other processes before the program exits. This section explains how to take advantage of this feature.

To intercept errors the application must give a callback routine to the OpenACC runtime. To provide the callback, the application calls `acc_set_error_routine` with a pointer to the callback routine.

The interface is the following, where `err_msg` contains a description of the error:

```
typedef void (*exitroutinetype)(char *err_msg);
extern void acc_set_error_routine(exitroutinetype callback_routine);
```

When the OpenACC runtime detects a runtime error, it will invoke the `callback_routine`.

> This feature is not the same as error recovery. If the callback routine returns to the application, the behavior is decidedly undefined.

Let's look at this feature in more depth using an example.

Take the MPI program below and run it with two processes. Process 0 tries to allocate a large array on the GPU, then sends a message to the second process to acknowledge the success of the operation. Process 1 waits for the acknowledgment and terminates upon receiving it.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define N 2147483648

int main(int argc, char **argv)
{
  int rank, size;
```

```
  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int ack;
  if(rank == 0) {
    float *a = (float*) malloc(sizeof(float) * N);

#pragma acc enter data create(a[0:N])
#pragma acc parallel loop independent
    for(int i = 0; i < N; i++) {
      a[i] = i *0.5;
    }
#pragma acc exit data copyout(a[0:N])
    printf("I am process %d, I have initialized a vector of size %ld bytes on
 the GPU. Sending acknowledgment to process 1.", rank, N);
    ack = 1;
    MPI_Send(&ack, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if(rank == 1) {
    MPI_Recv(&ack, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("I am process %d, I have received the acknowledgment from process 0
 that data in the GPU has been initialized.\n", rank, N);
    fflush(stdout);
  }

  // do some more work

  MPI_Finalize();

  return 0;
}
```

We compile the program with:

```
$ mpicc -ta=tesla -o error_handling_mpi error_handling_mpi.c
```

If we run this program with two MPI processes, the output will look like the following:

```
$ mpirun -n 2 ./error_handling_mpi
Out of memory allocating -8589934592 bytes of device memory
total/free CUDA memory: 11995578368/11919294464
Present table dump for device[1]:
NVIDIA Tesla GPU 0, compute capability 3.7, threadid=1
...empty...
call to cuMemAlloc returned error 2: Out of memory

--------------------------------------------------------
Primary job terminated normally, but 1 process returned
a non-zero exit code.. Per user-direction, the job has been aborted.
--------------------------------------------------------
--------------------------------------------------------------------------
mpirun detected that one or more processes exited with non-zero status,
thus causing the job to be terminated.
```

Process 0 failed while allocating memory on the GPU and terminated unexpectedly with an error. In this case mpirun was able to identify that one of the processes failed, so it shut down the remaining process and terminated the application. A simple two-process program like this is straightforward to debug. In a real world application though, with hundreds or thousands of processes, having a process exit prematurely may cause the application to hang indefinitely. Therefore it would be ideal to catch the failure of a process, control the termination of the other processes, and provide a useful error message.

We can use the OpenACC error handling feature to improve the previous program and correctly terminate the application in case of failure of an MPI process.

In the following sample code, we have added an error handling callback routine that will shut down the other processes if a process encounters an error while executing on the GPU. Process 0 tries to allocate a large array into the GPU and, if the operation is successful, process 0 will send an acknowledgment to process 1. Process 0 calls the OpenACC function `acc_set_error_routine` to set the function `handle_gpu_errors` as an error handling callback routine. This routine prints a message and calls `MPI_Abort` to shut down all the MPI processes. If process 0 successfully allocates the array on the GPU, process 1 will receive the acknowledgment. Otherwise, if process 0 fails, it will terminate itself and trigger the call to `handle_gpu_errors`. Process 1 is then terminated by the code executed in the callback routine.

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define N 2147483648


typedef void (*exitroutinetype)(char *err_msg);
extern void acc_set_error_routine(exitroutinetype callback_routine);

void handle_gpu_errors(char *err_msg) {
  printf("GPU Error: %s", err_msg);
  printf("Exiting...\n\n");
  MPI_Abort(MPI_COMM_WORLD, 1);
  exit(-1);
}


int main(int argc, char **argv)
{
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int ack;
  if(rank == 0) {
    float *a = (float*) malloc(sizeof(float) * N);


    acc_set_error_routine(&handle_gpu_errors);


#pragma acc enter data create(a[0:N])
#pragma acc parallel loop independent
    for(int i = 0; i < N; i++) {
      a[i] = i *0.5;
    }
#pragma acc exit data copyout(a[0:N])
    printf("I am process %d, I have initialized a vector of size %ld bytes on
 the GPU. Sending acknowledgment to process 1.", rank, N);
    fflush(stdout);
    ack = 1;
    MPI_Send(&ack, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if(rank == 1) {
```

```
    MPI_Recv(&ack, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("I am process %d, I have received the acknowledgment from process 0
 that data in the GPU has been initialized.\n", rank, N);
    fflush(stdout);
  }

  // more work

  MPI_Finalize();

  return 0;
}
```

Again, we compile the program with:

```
$ mpicc -ta=tesla -o error_handling_mpi error_handling_mpi.c
```

We run the program with two MPI processes and obtain the output below:

```
$ mpirun -n 2 ./error_handling_mpi
Out of memory allocating -8589934592 bytes of device memory
total/free CUDA memory: 11995578368/11919294464
Present table dump for device[1]:
NVIDIA Tesla GPU 0, compute capability 3.7, threadid=1
...empty...
GPU Error: call to cuMemAlloc returned error 2: Out of memory
Exiting...

--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode 1.
```

This time the error on the GPU was intercepted by the application which managed it with the error handling callback routine. In this case the routine printed some information about the problem and called `MPI_Abort` to terminate the remaining processes and avoid any unexpected behavior from the application.

# 2.11. C Examples

The simplest C example of OpenACC is a vector addition on the GPU:

```
#include <stdio.h>
#include <stdlib.h>
void vecaddgpu( float *restrict r, float *a, float *b, int n ){
    #pragma acc kernels loop copyin(a[0:n],b[0:n]) copyout(r[0:n])
    for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}

int main( int argc, char* argv[] ){
    int n; /* vector length */
    float * a; /* input vector 1 */
    float * b; /* input vector 2 */
    float * r; /* output vector */
    float * e; /* expected output values */
    int i, errs;
    if( argc > 1 ) n = atoi( argv[1] );
    else n = 100000; /* default vector length */
    if( n <= 0 ) n = 100000;
    a = (float*)malloc( n*sizeof(float) );
    b = (float*)malloc( n*sizeof(float) );
    r = (float*)malloc( n*sizeof(float) );
    e = (float*)malloc( n*sizeof(float) );
    for( i = 0; i < n; ++i ){
        a[i] = (float)(i+1);
```

```
        b[i] = (float)(1000*i);
    }
    /* compute on the GPU */
    vecaddgpu( r, a, b, n );
    /* compute on the host to compare */
    for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
    /* compare results */
    errs = 0;
    for( i = 0; i < n; ++i ){
        if( r[i] != e[i] ){
            ++errs;
        }
    }
    printf( "%d errors found\n", errs );
    return errs;
}
```

The important part of this example is the routine `vecaddgpu`, which includes one OpenACC directive for the loop. This (`#pragma acc`) directive tells the compiler to generate a kernel for the following loop (`kernels loop`), to allocate and copy from the host memory into the GPU memory n elements for the vectors a and b before executing on the GPU, starting at a[0] and b[0] (`copyin(a[0:n],b[0:n])`), and to allocate n elements for the vector r before executing on the GPU, and copy from the GPU memory out to the host memory those n elements, starting at r[0] (`copyout(r[0:n])`).

If you type this example into a file `a1.c`, you can build it with this release using the command `pgcc -acc a1.c`. The `-acc` flag enables recognition of the OpenACC pragmas and includes the OpenACC runtime library. This command generates the usual `a.out` executable file, and you run the program by running `a.out` as normal. You should see the output:

```
0 errors found
```

If instead you get the following output, then there is something wrong with your hardware installation or your GPU driver.

```
libcuda.so not found, exiting
Please check that the CUDA driver is installed and the shared object
is in the install directory or on your LD_LIBRARY_PATH.
```

You can enable additional output by setting environment variables. If you set the environment variable `PGI_ACC_NOTIFY` to 1, then the runtime prints a line of output each time you run a kernel on the GPU. For this program, you might get output that looks like:

```
launch CUDA kernel file=/user/guest/a1.c function=vecaddgpu
line=5 device=0 threadid=1 num_gangs=782 num_workers=1
vector_length=128 grid=782 block=128
0 errors found
```

The extra output tells you that the program launched a kernel for the loop at line 5, with a CUDA grid of size 782, and a thread block of size 128.

If you set the environment variable `PGI_ACC_NOTIFY` to 3, the output will include information about the data transfers as well:

```
upload CUDA data file=/user/guest/a1.c function=vecaddgpu
line=4 device=0 threadid=1 variable=a bytes=400000
upload CUDA data file=/user/guest/a1.c function=vecaddgpu
line=4 device=0 threadid=1 variable=b bytes=400000
launch CUDA kernel file=/user/guest/a1.c function=vecaddgpu
line=5 device=0 threadid=1 num_gangs=782 num_workers=1 vector_length=128
 grid=782 block=128
```

```
download CUDA data file=/user/guest/a1.c function=vecaddgpu
line=6 device=0 threadid=1 variable=r bytes=400000
0 errors found
```

If you set the environment variable `PGI_ACC_TIME` to 1, the runtime summarizes the time taken for data movement between the host and GPU, and computation on the GPU. On Linux, you may need to set the `LD_LIBRARY_PATH` environment variable to include the `/opt/pgi/linux86-64/19.10/lib` directory. This release dynamically loads a shared object to implement the profiling feature, and the path to the library must be available.

For this program, you might get output similar to this:

```
0 errors found

Accelerator Kernel Timing data
/user/guest/a1.c
  vecaddgpu  NVIDIA  devicenum=0
    time(us): 167
    4: compute region reached 1 time
        5: kernel launched 1 time
            grid: [782]  block: [128]
             device time(us): total=5 max=5 min=5 avg=5
            elapsed time(us): total=700 max=700 min=700 avg=700
    4: data region reached 2 times
        4: data copyin transfers: 2
             device time(us): total=110 max=67 min=43 avg=55
        6: data copyout transfers: 1
             device time(us): total=52 max=52 min=52 avg=52
```

This tells you that the program entered one accelerator region and spent a total of about 167 microseconds in that region. It copied two arrays to the device, launched one kernel, and brought one array back to the host.

You might also find it useful to enable the compiler feedback when you are writing your own OpenACC programs. This is enabled with the `-Minfo` flag. If you compile this program with the command `pgcc -acc -fast -Minfo a1.c`, you get the output:

```
 vecaddgpu:
      4, Generating copyin(a[:n])
         Generating copyout(r[:n])
         Generating copyin(b[:n])
      5, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          5, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      5, Loop not fused: no successor loop
         Generated 2 alternate versions of the loop
         Generated vector simd code for the loop
         Generated 2 prefetch instructions for the loop
         Generated vector simd code for the loop
         Generated 2 prefetch instructions for the loop
         Generated vector simd code for the loop
         Generated 2 prefetch instructions for the loop
main:
     21, Loop not fused: function call before adjacent loop
         Loop not vectorized: data dependency
         Loop unrolled 16 times
         Generated 1 prefetches in scalar loop
     28, Loop not fused: dependence chain to sibling loop
         Generated 2 alternate versions of the loop
         Generated vector and scalar versions of the loop; pointer conflict
 tests determine which is executed
         Generated 2 prefetch instructions for the loop
```

```
        Generated vector and scalar versions of the loop; pointer conflict
tests determine which is executed
        Generated 2 prefetch instructions for the loop
        Generated vector and scalar versions of the loop; pointer conflict
tests determine which is executed
        Generated 2 prefetch instructions for the loop
        Loop unrolled 16 times
        Generated 1 prefetches in scalar loop
    31, Loop not fused: function call before adjacent loop
```

This output gives the *schedule* used for the loop; in this case, the schedule is
`gang,vector(128)`. This means the iterations of the loop are broken into vectors of
128, and the vectors executed in parallel by SMs or compute units of the GPU.

This output is important because it tells you when you are going to get parallel
execution or sequential execution. If you remove the `restrict` keyword from the
declaration of the dummy argument *r* to the routine `vecaddgpu`, the `–Minfo` output
tells you that there may be dependences between the stores through the pointer r and
the fetches through the pointers a and b:

```
          5, Complex loop carried dependence of b->,a-> prevents
parallelization
        Loop carried dependence of r-> prevents parallelization
        Loop carried backward dependence of r-> prevents vectorization
        Accelerator serial kernel generated
        Accelerator kernel generated
        Generating Tesla code
         5, #pragma acc loop seq
      5, Complex loop carried dependence of b->,a-> prevents parallelization
        Loop carried dependence of r-> prevents parallelization
        Loop carried backward dependence of r-> prevents vectorization
        Loop not fused: no successor loop
        Generated 2 alternate versions of the loop
        Generated vector and scalar versions of the loop; pointer conflict
tests determine which is executed
        ...
```

A scalar kernel runs on one thread of one thread block, which runs about 1000 times
slower than the same parallel kernel. For this simple program, the total time is
dominated by GPU initialization, so you might not notice the difference in times, but in
production mode you need parallel kernel execution to get acceptable performance.

For our second example, we modify the program slightly by replacing the data clauses
on the kernels pragma with a `present` clause, and add a data construct surrounding
the call to the `vecaddgpu` routine. The data construct moves the data across to the GPU
in the main program. The `present` clause in the `vecaddgpu` routine tells the compiler
to use the GPU copy of the data that has already been allocated on the GPU. If you run
this program on the GPU with `PGI_ACC_TIME` set, you see that the kernel region now
has no data movement associated with it. Instead, the data movement is all associated
with the data construct in the main program.

```
#include <stdio.h>
#include <stdlib.h>

void vecaddgpu( float *restrict r, float *a, float *b, int n ){
        #pragma acc kernels loop present(r,a,b)
        for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}

int main( int argc, char* argv[] ){
        int n; /* vector length */
        float * a; /* input vector 1 */
```

```
        float * b; /* input vector 2 */
        float * r; /* output vector */
        float * e; /* expected output values */
        int i, errs;

        if( argc > 1 ) n = atoi( argv[1] );
        else n = 100000; /* default vector length */
        if( n <= 0 ) n = 100000;
        a = (float*)malloc( n*sizeof(float) );
        b = (float*)malloc( n*sizeof(float) );
        r = (float*)malloc( n*sizeof(float) );
        e = (float*)malloc( n*sizeof(float) );
        for( i = 0; i < n; ++i ){
            a[i] = (float)(i+1);
            b[i] = (float)(1000*i);
        }
        /* compute on the GPU */
        #pragma acc data copyin(a[0:n],b[0:n]) copyout(r[0:n])
        {
            vecaddgpu( r, a, b, n );
        }
        /* compute on the host to compare */
        for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
        /* compare results */
        errs = 0;
        for( i = 0; i < n; ++i ){
            if( r[i] != e[i] ){
            ++errs;
            }
        }
        printf( "%d errors found\n", errs );
        return errs;
}
```

# 2.12. Fortran Examples

The simplest Fortan example of OpenACC is a vector addition on the GPU.

## 2.12.1. Vector Addition on the GPU

The section contains two Fortan examples of vector addition on the GPU:

```
module vecaddmod
  implicit none
 contains
  subroutine vecaddgpu( r, a, b, n )
   real, dimension(:) :: r, a, b
   integer :: n
   integer :: i
!$acc kernels loop copyin(a(1:n),b(1:n)) copyout(r(1:n))
   do i = 1, n
    r(i) = a(i) + b(i)
   enddo
  end subroutine
end module

program main
  use vecaddmod
  implicit none
  integer :: n, i, errs, argcount
  real, dimension(:), allocatable :: a, b, r, e
  character*10 :: arg1
  argcount = command_argument_count()
  n = 1000000  ! default value
```

```
   if( argcount >= 1 )then
    call get_command_argument( 1, arg1 )
    read( arg1, '(i)' ) n
    if( n <= 0 ) n = 100000
   endif
   allocate( a(n), b(n), r(n), e(n) )
   do i = 1, n
    a(i) = i
    b(i) = 1000*i
   enddo
   ! compute on the GPU
   call vecaddgpu( r, a, b, n )
   ! compute on the host to compare
   do i = 1, n
    e(i) = a(i) + b(i)
   enddo
   ! compare results
   errs = 0
   do i = 1, n
    if( r(i) /= e(i) )then
      errs = errs + 1
    endif
   enddo
   print *, errs, ' errors found'
   if( errs ) call exit(errs)
end program
```

The important part of this example is the subroutine `vecaddgpu`, which includes one OpenACC directive for the loop. This (`!$acc`) directive tells the compiler to generate a kernel for the following loop (`kernels loop`), to allocate and copy from the host memory into the GPU memory n elements for the vectors a and b before executing on the GPU, starting at a(1) and b(1) (`copyin(a(1:n),b(1:n))`), and to allocate n elements for the vector r before executing on the GPU, and copy from the GPU memory out to the host memory those n elements, starting at r(1) (`copyout(r(1:n))`).

If you type this example into a file `f1.f90`, you can build it using the command `pgfortran -acc f1.f90`. The `-acc` flag enables recognition of the OpenACC pragmas and includes the OpenACC runtime library. This command generates the usual `a.out` executable file, and you run the program by running `a.out` as normal. You should see the output:

```
        0  errors found
```

If instead you get the following output, then there is something wrong with your hardware installation or your CUDA driver.

```
libcuda.so not found, exiting
Please check that the CUDA driver is installed and the shared object
is in the install directory or on your LD_LIBRARY_PATH.
```

You can enable additional output by setting environment variables. If you set the environment variable `PGI_ACC_NOTIFY` to 1, then the runtime prints a line of output each time you run a kernel on the GPU. For this program, you might get output that looks like:

```
launch CUDA kernel file=/user/guest/f1.f90 function=vecaddgpu
line=9 device=0 threadid=1 num_gangs=7813 num_workers=1
vector_length=128 grid=7813 block=128
        0  errors found
```

The extra output tells you that the program launched a kernel for the loop at line 9, with a CUDA grid of size 7813, and a thread block of size 128. If you set the environment

variable `PGI_ACC_NOTIFY` to 3, the output will include information about the data transfers as well:

```
upload CUDA data  file=/user/guest/f1.f90 function=vecaddgpu
line=8 device=0 threadid=1 variable=a bytes=4000000
upload CUDA data  file=/user/guest/f1.f90 function=vecaddgpu
line=8 device=0 threadid=1 variable=b bytes=4000000
launch CUDA kernel  file=/user/guest/f1.f90 function=vecaddgpu
line=9 device=0 threadid=1 num_gangs=7813 num_workers=1 vector_length=128
 grid=7813 block=128
download CUDA data  file=/user/guest/f1.f90 function=vecaddgpu
line=12 device=0 threadid=1 variable=r bytes=4000000
            0  errors found
```

If you set the environment variable `PGI_ACC_TIME` to 1, the runtime summarizes the time taken for data movement between the host and GPU, and computation on the GPU. For this program, you might get output similar to this:

```
            0  errors found

Accelerator Kernel Timing data
/home/ams/tat/example-f/f1.f90
  vecaddgpu  NVIDIA  devicenum=0
    time(us): 1,040
    8: compute region reached 1 time
        9: kernel launched 1 time
            grid: [7813]  block: [128]
             device time(us): total=19 max=19 min=19 avg=19
             elapsed time(us): total=738 max=738 min=738 avg=738
    8: data region reached 2 times
        8: data copyin transfers: 2
            device time(us): total=689 max=353 min=336 avg=344
        12: data copyout transfers: 1
            device time(us): total=332 max=332 min=332 avg=332
```

This tells you that the program entered one accelerator region and spent a total of about 1 millisecond in that region. It copied two arrays to the device, launched one kernel and brought one array back to the host.

You might also find it useful to enable the compiler feedback when you are writing your own OpenACC programs. This is enabled with the `-Minfo` flag.

If you compile this program with the command `pgfortran -acc -fast -Minfo f1.f90`, you get the output:

```
vecaddgpu:
      8, Generating copyin(a(:n))
         Generating copyout(r(:n))
         Generating copyin(b(:n))
      9, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          9, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
      9, Loop not fused: no successor loop
         Generated 2 alternate versions of the loop
         Generated vector simd code for the loop
         Generated 2 prefetch instructions for the loop
         Generated vector simd code for the loop
         Generated 2 prefetch instructions for the loop
         Generated vector simd code for the loop
         Generated 2 prefetch instructions for the loop
main:
     29, Loop not fused: function call before adjacent loop
         Loop not vectorized: may not be beneficial
         Unrolled inner loop 8 times
     36, Loop not fused: function call before adjacent loop
```

```
      2 loops fused
```

This output gives the schedule used for the loop; in this case, the schedule is gang, vector(128). This means the iterations of the loop are broken into vectors of 128, and the vectors are executed in parallel by SMs of the GPU. This output is important because it tells you when you are going to get parallel execution or sequential execution.

For our second example, we modify the program slightly by replacing the data clauses on the kernels pragma with a present clause, and add a data construct surrounding the call to the vecaddgpu subroutine. The data construct moves the data across to the GPU in the main program. The present clause in the vecaddgpu subroutine tells the compiler to use the GPU copy of the data that has already been allocated on the GPU. If you run this program on the GPU with PGI_ACC_TIME set, you see that the kernel region now has no data movement associated with it. Instead, the data movement is all associated with the data construct in the main program.

In Fortran programs, you don't have to specify the array bounds in data clauses if the compiler can figure out the bounds from the declaration, or if the arrays are assumed-shape dummy arguments or allocatable arrays.

```fortran
module vecaddmod
  implicit none
 contains
  subroutine vecaddgpu( r, a, b, n )
   real, dimension(:) :: r, a, b
   integer :: n
   integer :: i
!$acc kernels loop present(r,a,b)
   do i = 1, n
    r(i) = a(i) + b(i)
   enddo
  end subroutine
end module

program main
  use vecaddmod
  implicit none
  integer :: n, i, errs, argcount
  real, dimension(:), allocatable :: a, b, r, e
  character*10 :: arg1
  argcount = command_argument_count()
  n = 1000000  ! default value
  if( argcount >= 1 )then
  call get_command_argument( 1, arg1 )
   read( arg1, '(i)' ) n
   if( n <= 0 ) n = 100000
  endif
  allocate( a(n), b(n), r(n), e(n) )
  do i = 1, n
   a(i) = i
   b(i) = 1000*i
  enddo
  ! compute on the GPU
!$acc data copyin(a,b) copyout(r)
  call vecaddgpu( r, a, b, n )
!$acc end data
  ! compute on the host to compare
  do i = 1, n
   e(i) = a(i) + b(i)
  enddo
  ! compare results
  errs = 0
  do i = 1, n
```

```
  if( r(i) /= e(i) )then
     errs = errs + 1
  endif
 enddo
 print *, errs, ' errors found'
 if( errs ) call exit(errs)
end program
```

## 2.12.2. Multi-Threaded Program Utilizing Multiple Devices

This simple example shows how to run a multi-threaded host program that utilizes multiple devices.

```
program tdot
! Compile with "pgfortran -mp -acc tman.f90 -lblas
! Set OMP_NUM_THREADS environment variable to run with
! up to 2 threads, currently.
!
use openacc
use omp_lib
!
integer, parameter :: N = 10000
real*8 x(N), y(N), z
integer, allocatable :: offs(:)
real*8, allocatable :: zs(:)
real*8 ddot

! Max at 2 threads for now
nthr = omp_get_max_threads()
if (nthr .gt. 2) nthr = 2
call omp_set_num_threads(nthr)

! Run on host
call random_number(x)
call random_number(y)
z = ddot(N,x,1,y,1)
print *,"Host Serial",z

! Attach each thread to a device
!$omp PARALLEL private(i)
    i = omp_get_thread_num()
    call acc_set_device_num(i, acc_device_nvidia)
!$omp end parallel

! Break up the array into sections
nsec = N / nthr
allocate(offs(nthr),zs(nthr))
offs = (/ (i*nsec,i=0,nthr-1) /)
zs = 0.0d0

! Decompose the problem across devices
!$omp PARALLEL private(i,j,z)
    i = omp_get_thread_num() + 1
    z = 0.0d0
    !$acc kernels loop &
       copyin(x(offs(i)+1:offs(i)+nsec),y(offs(i)+1:offs(i)+nsec))
    do j = offs(i)+1, offs(i)+nsec
       z = z + x(j) * y(j)
    end do
    zs(i) = z
!$omp end parallel
z = sum(zs)
print *,"Multi-Device Parallel",z
end
```

The program starts by having each thread call `acc_set_device_num` so each thread will use a different GPU. Within the computational OpenMP parallel region, each thread copies the data it needs to its GPU and proceeds.

# 2.13. Troubleshooting Tips and Known Limitations

This release of the PGI compilers does not implement the full OpenACC specification. For an explanation of what features are not yet implemented, refer to Chapter 3, Implemented Features.

The Linux CUDA driver will power down an idle GPU. This means if you are using a GPU with no attached display, or an NVIDIA Tesla compute-only GPU, and there are no open CUDA contexts, the GPU will power down until it is needed. Since it may take up to a second to power the GPU back up, you may experience noticeable delays when you start your program. When you run your program with the environment variable `PGI_ACC_TIME` set to 1, this time will appear as initialization time. If you are running many tests, or want to isolate the actual time from the initialization time, you can run the PGI utility `pgcudainit` in the background. This utility opens a CUDA context and holds it open until you kill it or let it complete.

This release has support for the `async` clause and `wait` directive. When you use asynchronous computation or data movement, you are responsible for ensuring that the program has enough synchronization to resolve any data races between the host and the GPU. If your program uses the `async` clause and wrong answers are occuring, you can test whether the `async` clause is causing problems by setting the environment variable `PGI_ACC_SYNCHRONOUS` to 1 before running your program. This action causes the OpenACC runtime to ignore the `async` clauses and run the program in synchronous mode.

# Chapter 3.
# IMPLEMENTED FEATURES

This section lists the OpenACC features available in this release, and the features to be implemented in upcoming PGI releases.

## 3.1. In This Release

This release includes full support for the OpenACC 2.5 specification except for the following features:

- ▸ Declare link
- ▸ Nested parallelism
- ▸ Restricting **cache** clause variable refs to variables within a cached region.

## 3.2. Defaults

In this release, the default `ACC_DEVICE_TYPE` is `acc_device_nvidia`, just as the `-acc` compiler option targets `-ta=tesla` by default. The device types `acc_device_default` and `acc_device_not_host` behave the same as `acc_device_nvidia`. The device type can be changed using the environment variable or by a call to `acc_set_device_type()`.

In this release, the default `ACC_DEVICE_NUM` is 0 for the `acc_device_nvidia` type, which is consistent with the CUDA device numbering system. For more information, refer to the `pgaccelinfo` output in Prepare Your System. The device number can be changed using the environment variable or by a call to `acc_set_device_num`.

## 3.3. Environment Variables

This section summarizes the environment variables that PGI OpenACC supports. These environment variables are user-setable environment variables that control behavior of accelerator-enabled programs at execution. These environment variables must comply with these rules:

- ▸ The names of the environment variables must be upper case.
- ▸ The values of 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.

The following table contains the environment variables that are currently supported and provides a brief description of each.

## Table 1   Supported Environment Variables

| Use this environment variable... | To do this... |
|---|---|
| PGI_ACC_BUFFERSIZE | For NVIDIA CUDA devices, this defines the size of the pinned buffer used to transfer data between host and device. |
| PGI_ACC_CUDA_PROFSTOP | Set to 1 (or any positive value) to tell the PGI runtime environment to insert an 'atexit(cuProfilerStop)' call upon exit. This behavior may be desired in the case where a profile is incomplete or where a message is issued to call cudaProfilerStop(). |
| PGI_ACC_DEBUG | Set to 1 to instruct the PGI runtime to generate information about device memory allocation, data movement, kernel launches, and more. PGI_ACC_DEBUG is designed mostly for use in debugging the runtime itself, but it may be helpful in understanding how the program interacts with the device. Expect copious amounts of output. |
| PGI_ACC_DEVICE_NUM  = = ACC_DEVICE_NUM | Sets the default device number to use. PGI_ACC_DEVICE_NUM overrides ACC_DEVICE_NUM. 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. |
| PGI_ACC_DEVICE_TYPE  = = ACC_DEVICE_TYPE | Sets the default device type to use. PGI_ACC_DEVICE_TYPE overrides ACC_DEVICE_TYPE. Controls which accelerator device 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, and currently may be the string NVIDIA, TESLA, or HOST |
| PGI_ACC_GANGLIMIT | For NVIDIA CUDA devices, this defines the maximum number of gangs (CUDA thread blocks) that will be launched by a kernel. |
| PGI_ACC_NOTIFY | Writes out a line for each kernel launch and/or data movement. When set to an integer value, the value, is used as a bit mask to print information about kernel launches (value 1), data transfers (value 2), region entry/exit (value 4), wait operations or synchronizations with the device (value 8), and device memory allocates and deallocates (value 16). |
| PGI_ACC_PROFLIB | Enables 3rd party tools interface using the new profiler dynamic library interface. |
| PGI_ACC_SYNCHRONOUS | Disables asynchronous launches and data movement. |
| PGI_ACC_TIME | Enables a lightweight profiler to measure data movement and accelerator kernel execution time and print a summary at the end of program execution. |

# 3.4. OpenACC Fortran API Extensions

This section summarizes the OpenACC 2.0 Fortran API extensions that PGI supports.

## 3.4.1. acc_malloc

The `acc_malloc` function returns a device pointer, in a variable of type(c_devptr), to newly allocated memory on the device. If the data can not be allocated, this function returns C_NULL_DEVPTR.

There is one supported call format in PGI Fortran:

```
type(c_devptr) function acc_malloc (bytes)
```

where *bytes* is an integer which specifies the number of bytes requested.

## 3.4.2. acc_free

The `acc_free` subroutine frees memory previously allocated by `acc_malloc`. It takes as an argument either a device pointer contained in an instance of derived type(c_devptr), or for convenience, a CUDA Fortran device array. In PGI Fortran, calling `acc_free` (or `cudaFree`) with a CUDA Fortran device array that was allocated using the F90 allocate statement results in undefined behavior.

There are two supported call formats in PGI Fortran:

```
subroutine acc_free ( devptr )
```

where *devptr* is an instance of derived type(c_devptr)

```
subroutine acc_free ( dev )
```

where *dev* is a CUDA Fortran device array

## 3.4.3. acc_map_data

The `acc_map_data` routine associates (maps) host data to device data. The first argument is a host array, contiguous host array section, or address contained in a type(c_ptr). The second argument must be a device address contained in a type(c_devptr), such as would be returned from `acc_malloc` or `acc_deviceptr`, or a CUDA Fortran device array.

There are four supported call formats in PGI Fortran:

```
subroutine acc_map_data ( host, dev, bytes )
```

where *host* is a host variable, array or starting array element
*dev* is a CUDA Fortran device variable, array, or starting array element
*bytes* is an integer which specifies the mapping length in bytes)

```
subroutine acc_map_data ( host, dev )
```

where *host* is a host array or contiguous host array section

*dev* is a CUDA Fortran device array or array section which conforms to host

```
subroutine acc_map_data ( host, devptr, bytes )
```

where *host* is a host variable, array or starting array element

*devptr* is an instance of derived type(c_devptr)

*bytes* is an integer which specifies the mapping length in bytes)

```
subroutine acc_map_data ( ptr, devptr, bytes )
```

where *ptr* is an instance of derived type(c_ptr)

*devptr* is an instance of derived type(c_devptr)

*bytes* is an integer which specifies the mapping length in bytes)

## 3.4.4. acc_unmap_data

The `acc_unmap_data` routine unmaps (or disassociates) the device data from the specified host data.

There is one supported call format in PGI Fortran:

```
subroutine acc_unmap_data ( host )
```

where *host* is a host variable that was mapped to device data in a previous call to acc_map_data.

## 3.4.5. acc_deviceptr

The `acc_deviceptr` function returns the device pointer, in a variable of type(c_devptr), mapped to a host address. The input argument is a host variable or array element that has an active lifetime on the current device. If the data is not present, this function returns C_NULL_DEVPTR.

There is one supported call format in PGI Fortran:

```
type(c_devptr) function acc_deviceptr ( host )
```

where *host* is a host variable or array element of any type, kind and rank.

## 3.4.6. acc_hostptr

The `acc_hostptr` function returns the host pointer, in a variable of type(c_ptr), mapped to a device address. The input argument is a device address, such as would be returned from acc_malloc or acc_deviceptr, or a CUDA Fortran device array.

There are two supported call formats in PGI Fortran:

```
type(c_ptr) function acc_hostptr ( dev )
```

where *dev* is a CUDA Fortran device array

```
type(c_ptr) function acc_hostptr ( devptr )
```

where *devptr* is an instance of derived type(c_devptr)

### 3.4.7. acc_is_present

The `acc_is_present` function returns .true. or .false. depending on whether a host variable or array region is present on the device.

There are two supported call formats in PGI Fortran:

```
logical function acc_is_present ( host )
```

where *host* is a contiguous array section of intrinsic type.

```
logical function acc_is_present ( host, bytes )
```

where *host* is a host variable of any type, kind, and rank.
*bytes* is an integer which specifies the length of the data to check.

### 3.4.8. acc_memcpy_to_device

The `acc_memcpy_to_device` routine copies data from local memory to device memory. The source address is a host array, contiguous array section, or address contained in a type(c_ptr). The destination address must be a device address, such as would be returned from `acc_malloc` or `acc_deviceptr`, or a CUDA Fortran device array.

There are four supported call formats in PGI Fortran:

```
subroutine acc_memcpy_to_device ( dev, src, bytes )
```

where *dev* is a CUDA Fortran device variable, array or starting array element.
*src* is a host variable, array, or starting array element.
*bytes* is an integer which specifies the length of the copy in bytes.

```
subroutine acc_memcpy_to_device ( dev, src )
```

where *dev* is a CUDA Fortran device array or contiguous array section.
*src* is a host array or array section which conforms to dev.

```
subroutine acc_memcpy_to_device ( devptr, src, bytes )
```

where *devptr* is an instance of derived type(c_devptr).
*src* is a host variable, array, or starting array element.
*bytes* is an integer which specifies the length of the copy in bytes.

```
subroutine acc_memcpy_to_device ( devptr, ptr, bytes )
```

where *devptr* is an instance of derived type(c_devptr).
*ptr* is an instance of derived type(c_ptr).
*bytes* is an integer which specifies the length of the copy in bytes.

### 3.4.9. acc_memcpy_from_device

The `acc_memcpy_from_device` routine copies data from device memory to local memory. The source address must be a device address, such as would be returned from `acc_malloc`, `acc_deviceptr`, or a CUDA Fortran device array. The source address is a host array, contiguous array section, or address contained in a type(c_ptr).

There are four supported call formats in PGI Fortran:

```
subroutine acc_memcpy_from_device ( dest, dev, bytes )
```

where *dest* is a host variable, array, or starting array element.
*dev* is a CUDA Fortran device variable, array or starting array element.
*bytes* is an integer which specifies the length of the copy in bytes)

```
subroutine acc_memcpy_from_device ( dest, dev )
```

where *dest* is a host array or contiguous array section.
*dev* is a CUDA Fortran device array or array section which conforms to dest
subroutine.

```
subroutine acc_memcpy_from_device ( dest, devptr, bytes )
```

where *dest* is a host variable, array, or starting array element.
*devptr* is an instance of derived type(c_devptr).
*bytes* is an integer which specifies the length of the copy in bytes)

```
subroutine acc_memcpy_from_device ( ptr, devptr, bytes )
```

where *ptr* is an instance of derived type(c_ptr).
*devptr* is an instance of derived type(c_devptr).
*bytes* is an integer which specifies the length of the copy in bytes)

## 3.4.10. acc_get_cuda_stream

The `acc_get_cuda_stream` function returns the CUDA stream value which
corresponds to an OpenACC async queue. The input argument is an async number or
a pre-defined value such as acc_async_sync. This call is only supported on NVIDIA
platforms.

There is one supported call format in PGI Fortran:

```
integer(acc_handle_kind) function acc_get_cuda_stream ( async )
```

where *async* is a user-defined or pre-defined async value.

## 3.4.11. acc_set_cuda_stream

The `acc_set_cuda_stream` subroutine sets the CUDA stream value for an OpenACC
async queue on the current device. The input arguments are an async number and a
stream. This call is only supported on NVIDIA platforms.

There is one supported call format in PGI Fortran:

```
subroutine acc_set_cuda_stream ( async, stream )
```

where *async* and *stream* are integers of acc_handle_kind.

## 3.5. Known Limitations

This section includes the known limitations to OpenACC directives. PGI plans to
support these features in a future release.

## 3.5.1. ACC routine directive Limitations

▸ Extern variables may not be used with `acc routine` procedures.
▸ Reductions in procedures with `acc routine` are only supported for NVIDIA GPUs supporting compute capability 3.0 or higher.
▸ Fortran assumed-shape arguments are not yet supported.

## 3.5.2. C++ and OpenACC Limitations

There are limitations to the data that can appear in OpenACC data constructs and compute regions:

▸ Variable-length arrays are not supported in OpenACC data clauses; VLAs are not part of the C++ standard.
▸ Variables of class type that require constructors and destructors do not behave properly when they appear in data clauses.
▸ Exceptions are not handled in compute regions.
▸ Member variables are not fully supported in the use_device clause of a host_data construct; this placement may result in an error at runtime.

Conflicts may arise between the version of GCC required to enable C++ language feature support (GCC 5 or newer for C++14, GCC 7 or newer for C++17) and use of the `nonvvm` sub-option to `-ta=tesla`. The `nonvvm` sub-option uses components of the CUDA Toolkit which check for compatibility with the version of GCC installed on a system and will not work with a version newer than their specified maximum.

Table 2  GCC Version Compatibility with -ta=tesla:nonvvm

| CUDA Toolkit Version | Maximum GNU Version Supported |
|---|---|
| CUDA 9.2 | GCC 7.x |
| CUDA 10.0 | GCC 7.x |
| CUDA 10.1 | GCC 8.x |

## 3.5.3. Known Limitations

▸ This release does not support targeting another accelerator device after `acc_shutdown` has been called.

# 3.6. Interactions with Optimizations

This section discusses interactions with compiler optimizations that programmers should be aware of.

## 3.6.1. Interactions with Inlining

Procedure inlining may be enabled in several ways. User-controlled inlining is enabled using the `-Minline` flag, or with `-Mextract=lib:` and `-Minline=lib:` flags. For C and C++, compiler-controlled inlining is enabled using the `-Mautoinline` or `-fast` flags. Interprocedural analysis can also control inlining using the `-Mipa=inline` option. Inlining is a performance optimization by removing the overhead of the procedure call, and by specializing and optimizing the code of the inlined procedure at the point of the call site.

> As of the PGI 16.3 release, `-Mipa` has been disabled on Windows.

When a procedure containing a compute construct (`acc parallel` or `acc kernels`) is inlined into an `acc data` construct, the compiler will use the data construct clauses to optimize data movement between the host and device. In some cases, this can produce different answers, when the host and device copies of some variable are different. For instance, the data construct may specify a data clause for a scalar variable or a Fortran common block that contains a scalar variable. The compute construct in the inlined procedure will now see that the scalar variable is present on the device, and will use the device copy of that variable. Before inlining, the compute construct may have used the default `firstprivate` behavior for that scalar variable, which would use the host value for the variable.

- The `wait` clause on OpenACC directives is not supported.
- The `async` clause on the `wait` directive is not supported.
- The `device_type` clause is not supported on any directive.

## 3.7. In Future Releases

The following OpenACC features are not implemented in this release. They will be in future releases.

- Declare link
- Nested parallelism
- Restricting **cache** clause variable refs to variables within cached region.

# Chapter 4.
# CONTACT INFORMATION

You can contact NVIDIA's PGI compilers and tools team at:

9030 NE Walker Road, Suite 100
Hillsboro, OR 97006

Or electronically using any of the following means:

Fax: +1-503-682-2637
Sales: sales@pgroup.com
WWW: https://www.pgroup.com or pgicompilers.com

The PGI User Forum, pgicompilers.com/userforum is monitored by members of the PGI engineering and support teams as well as other PGI customers. The forums contain answers to many commonly asked questions. Log in to the PGI website, pgicompilers.com/login to access the forums.

Many questions and problems can be resolved by following instructions and the information available in the PGI frequently asked questions (FAQ), pgicompilers.com/faq.

Submit support requests using the PGI Technical Support Request form, pgicompilers.com/support-request.