

PGI Release Notes

Version 2014



PGI Compilers and Tools

TABLE OF CONTENTS

Chapter 1. Release Overview.....	1
1.1. Product Overview.....	1
1.1.1. Licensing Terminology.....	1
1.1.2. Licensing Options.....	2
1.1.3. PGI Workstation and PGI Server Comparison.....	2
1.1.4. PGI CDK Cluster Development Kit.....	2
1.2. Release Components.....	2
1.2.1. Additional Components for PGI CDK.....	3
1.2.2. MPI Support.....	3
1.3. Terms and Definitions.....	3
1.4. Supported Platforms.....	4
1.5. Supported Operating System Updates.....	4
1.5.1. Linux.....	4
1.5.2. OS X.....	4
1.5.3. Windows.....	5
1.6. Getting Started.....	5
Chapter 2. New and Modified Features.....	7
2.1. What's New in Release 2014.....	7
2.2. New and Modified Compiler Options.....	11
2.2.1. Required Suboption.....	11
2.2.2. Accelerator Options.....	11
2.2.3. Relocatable Device Code.....	14
2.2.4. LLVM/SPIR and Native GPU Code Generation.....	14
2.2.5. DWARF Debugging Formats.....	14
2.2.6. -tp Modifications.....	15
2.3. New and Modified Fortran Functionality.....	15
2.3.1. Contiguous Pointers.....	15
2.4. New and Modified Tools Functionality.....	15
2.5. Using MPI.....	16
2.6. PGI Accelerator Enhancements.....	17
2.6.1. OpenACC Directive Summary.....	17
2.6.2. CUDA Toolkit Version.....	19
2.6.3. C Structs in OpenACC.....	19
2.6.4. C++ Classes in OpenACC.....	21
2.6.5. Fortran Derived Types in OpenACC.....	25
2.6.6. OpenACC Atomic Support.....	27
2.6.7. OpenACC declare data directive for global and Fortran module variables.....	28
2.7. C++ Compiler.....	30
2.7.1. C++ and OpenACC.....	30
2.7.2. C++ Compatibility.....	31

2.8. New and Modified Runtime Library Routines.....	31
2.9. Library Interfaces.....	31
2.10. Environment Modules.....	31
Chapter 3.Distribution and Deployment.....	32
3.1. Application Deployment and Redistributables.....	32
3.1.1. PGI Redistributables.....	32
3.1.2. Linux Redistributables.....	32
3.1.3. Microsoft Redistributables.....	33
Chapter 4.Troubleshooting Tips and Known Limitations.....	34
4.1. General Issues.....	34
4.2. Platform-specific Issues.....	34
4.2.1. Linux.....	35
4.2.2. Apple OS X.....	35
4.2.3. Microsoft Windows.....	35
4.3. PGDBG-related Issues.....	36
4.4. PGPROF-related Issues.....	36
4.5. CUDA Toolkit Issues.....	36
4.6. OpenACC Issues.....	37
4.7. Corrections.....	37
Chapter 5.Contact Information.....	38

LIST OF TABLES

Table 1	Typical <code>-fast</code> and <code>-fastsse</code> Options	5
Table 2	Additional <code>-fast</code> and <code>-fastsse</code> Options	6
Table 3	<code>-ta=tesla</code> Suboptions	12
Table 4	<code>-ta=radeon</code> Suboptions	13
Table 5	MPI Distribution Options	16

Chapter 1.

RELEASE OVERVIEW

Welcome to Release 2014 of PGI Workstation™, PGI Server™, and the PGI CDK® Cluster Development Kit, a set of compilers and development tools for 32-bit and 64-bit x86-compatible processor-based workstations, servers, and clusters running versions of the Linux operating system. PGI Workstation and PGI Server are also available for the Apple OS X and Microsoft Windows operating systems.

This document describes changes between previous versions of the PGI 2014 release as well as late-breaking information not included in the current printing of the PGI Compiler User's Guide.

1.1. Product Overview

PGI Workstation, PGI Server, and the PGI CDK include exactly the same PGI compiler and tools software. The difference is the manner in which the license keys enable the software.

1.1.1. Licensing Terminology

The PGI compilers and tools are license-managed. Before discussing licensing, it is useful to have common terminology.

- ▶ **License** - a legal agreement between NVIDIA and PGI end-users, to which users assent upon installation of any PGI product. The terms of the License are kept up-to-date in documents on pgroup.com and in the \$PGI/<platform>/<rel_number> directory of every PGI software installation.
- ▶ **License keys** - ASCII text strings that enable use of the PGI software and are intended to enforce the terms of the License. License keys are generated by each PGI end-user on pgroup.com using a unique hostid and are typically stored in a file called `license.dat` that is accessible to the systems for which the PGI software is licensed.
- ▶ **PIN** - Personal Identification Number, a unique 6-digit number associated with a license. This PIN is included in your PGI order confirmation. The PIN can also be found in your PGI license file after **VENDOR_STRING=**.
- ▶ **License PIN code** - A unique 16-digit number associated with each PIN that enables users to "tie" that PIN to their pgroup.com user account. This code is provided by PIN owners to others whom they wish tied to their PIN(s).

1.1.2. Licensing Options

PGI offers licenses for either x64+accelerator or x64 only platforms. PGI Accelerator™ products, the x64+accelerator platform products, include support for the directive-based OpenACC programming model, CUDA Fortran and PGI CUDA-x86. PGI Accelerator compilers are supported on all Intel and AMD x64 processor-based systems with either CUDA-enabled NVIDIA GPUs or select AMD GPUs and APUs, running Linux, OS X, or Windows. OS X accelerator support is available only on NVIDIA GPUs.

1.1.3. PGI Workstation and PGI Server Comparison

- ▶ All PGI Workstation products include a node-locked single-user license, meaning one user at a time can compile on the one system on which the PGI Workstation compilers and tools are installed. The product and license server are on the same local machine.
- ▶ PGI Server products are offered in configurations identical to PGI Workstation, but include network-floating multi-user licenses. This means that two or more users can use the PGI compilers and tools concurrently on any compatible system networked to the license server, that is, the system on which the PGI Server license keys are installed. There can be multiple installations of the PGI Server compilers and tools on machines connected to the license server; and the users can use the product concurrently, provided they are issued a license key by the license server.

1.1.4. PGI CDK Cluster Development Kit

A cluster is a collection of compatible computers connected by a network. The PGI CDK supports parallel computation on clusters of 64-bit x86-compatible AMD and Intel processor-based Linux workstations or servers with or without accelerators and interconnected by a TCP/IP-based network, such as Ethernet.

Support for cluster programming does not extend to clusters combining 64-bit processor-based systems with 32-bit processor-based systems.

1.2. Release Components

Release 2014 includes the following components:

- ▶ PGFORTRAN™ native OpenMP and auto-parallelizing Fortran 2003 compiler.
- ▶ PGCC® native OpenMP and auto-parallelizing ANSI C99 and K&R C compiler.
- ▶ PGC++® native OpenMP and auto-parallelizing ANSI C++ compiler.
- ▶ PGPROF® MPI, OpenMP, and multi-thread graphical profiler.
- ▶ PGDBG® MPI, OpenMP, and multi-thread graphical debugger.
- ▶ MPICH MPI libraries, version 3.0.4, for 64-bit development environments (Linux and OS X only).



64-bit linux86-64 MPI messages are limited to <2GB size each.

- ▶ Microsoft HPC Pack 2012 MS-MPI Redistributable Pack (version 4.1) for 64-bit and 32-bit development environments (Windows only).
- ▶ LAPACK linear algebra math library for shared-memory vector and parallel processors, version 3.4.2, supporting Level 3 BLACS (Basic Linear Algebra Communication Subroutines) for use with PGI compilers. This library is provided in both 64-bit and 32-bit versions for AMD64 or Intel 64 CPU-based installations running Linux, OS X, or Windows.
- ▶ ScaLAPACK 2.0.2 linear algebra math library for distributed-memory systems for use with MPICH, Open MPI, MVAPICH, and the PGI compilers on 64-bit Linux and OS X for AMD64 or Intel 64 CPU-based installations.
- ▶ A UNIX-like shell environment for 32-bit and 64-bit Windows platforms.
- ▶ FlexNet license utilities.
- ▶ Documentation in PDF and man page formats.

1.2.1. Additional Components for PGI CDK

The PGI CDK for Linux includes additional components available for download from the PGI website, but not contained in the installation package:

- ▶ MVAPICH2 MPI libraries, version 1.9 available for 64-bit development environments.
- ▶ Open MPI libraries, version 1.7.3 for 64-bit development environments.

1.2.2. MPI Support

You can use PGI products to develop, debug, and profile MPI applications. The PGPROF[®] MPI profiler and PGDBG[®] debugger included with PGI Workstation are limited to eight local processes. The versions included with PGI Server are limited to 16 local processes. The MPI profiler and debugger included with PGI CDK supports up to 64 or 256 remote processes, depending on the purchased capabilities.

1.3. Terms and Definitions

This document contains a number of terms and definitions with which you may or may not be familiar. If you encounter an unfamiliar term in these notes, please refer to the online glossary at <http://www.pgroup.com/support/definitions.htm>

These two terms are used throughout the documentation to reflect groups of processors:

AMD64

A 64-bit processor from AMD[™] designed to be binary compatible with 32-bit x86 processors, and incorporating new features such as additional registers and 64-bit addressing support for improved performance and greatly increased memory range. This term includes the AMD Athlon64[™], AMD Opteron[™], AMD Turion[™], AMD Barcelona, AMD Shanghai, AMD Istanbul, AMD Bulldozer, and AMD Piledriver processors.

Intel 64

A 64-bit IA32 processor with Extended Memory 64-bit Technology extensions designed to be binary compatible with AMD64 processors. This includes Intel Pentium 4, Intel Xeon, Intel

Core 2, Intel Core 2 Duo (Penryn), Intel Core (i3, i5, i7), both first generation (Nehalem) and second generation (Sandy Bridge) processors, as well as Ivy Bridge and Haswell processors.

1.4. Supported Platforms

There are six platforms supported by the PGI Workstation and PGI Server compilers and tools. Currently, PGI CDK supports only 64-bit Linux clusters.

- ▶ **32-bit Linux** — includes all features and capabilities of the 32-bit Linux operating systems running on an x64 compatible processor. 64-bit Linux compilers will not run on these systems.
- ▶ **64-bit Linux** — includes all features and capabilities of the 64-bit Linux operating systems running on an x64 compatible processor. Both 64-bit and 32-bit Linux compilers run on these systems.
- ▶ **32-bit Windows** — includes all features of the 32-bit Windows operating systems running on either a 32-bit x86 compatible or an x64 compatible processor. 64-bit Windows compilers will not run on these systems.
- ▶ **64-bit Windows** — includes all features and capabilities of the 64-bit Windows version running on an x64 compatible processor. Both 64-bit and 32-bit Windows compilers run on these systems.
- ▶ **32-bit OS X** supported on 32-bit Apple operating systems running on either a 32-bit or 64-bit Intel-based Mac system. 64-bit OS X compilers will not run on these systems.
- ▶ **64-bit OS X** — supported on 64-bit Apple operating systems running on a 64-bit Intel-based Mac system. Both 64-bit and 32-bit OS X compilers run on these systems.

1.5. Supported Operating System Updates

This section describes updates and changes to PGI 2014 that are specific to Linux, OS X, and Windows.

1.5.1. Linux

Linux download packages are reorganized in PGI 2014. You can download a 64-bit Linux compiler package for installation on 64-bit Linux machines, and/or you can download a 32-bit package that installs on 32-bit and 64-bit Linux systems.

- ▶ RHEL 4.8+, including RHEL 6.5
- ▶ Fedora 4+, including Fedora 20
- ▶ SuSE 9.3+, including SuSE 13.1
- ▶ SLES 10+, including SLES 11 SP 3
- ▶ Ubuntu 8.04+, including Ubuntu 13.10

1.5.2. OS X

PGI 2014 for OS X supports most of the features of the 32-bit and 64-bit versions for linux86 and linux86-64 environments. Except where noted in these release notes or the user manuals, the PGI compilers and tools on OS X function identically to their Linux counterparts.

- ▶ Supported versions are OS X versions 10.6 (Snow Leopard) and newer, including 10.9 (Mavericks).

1.5.3. Windows

PGI 2014 for Windows supports most of the features of the 32-bit and 64-bit versions for linux86 and linux86-64 environments.



Starting January 2015, PGI releases will no longer include support for Windows XP, Windows Server 2003, or Windows Server 2008.

These releases are supported in PGI 2014, and require that the Microsoft Windows 8.1 Software Development Kit (SDK) be installed prior to installing the compilers.

- ▶ Windows Server 2008 R2
- ▶ Windows 7
- ▶ Windows 8
- ▶ Windows 8.1
- ▶ Windows Server 2012

PGI products on all Windows systems include Microsoft Open Tools. On the systems being deprecated in 2015, it contains all the tools needed for building executables. On newer Windows systems, Open Tools also needs the SDK to build executables.



PGI 2014 requires the 8.1 SDK, even on Windows 7 and Windows 8. The Windows 8 SDK requires the PGI 2013 release.

1.6. Getting Started

By default, the PGI 2014 compilers generate code that is optimized for the type of processor on which compilation is performed, the compilation host. If you are unfamiliar with the PGI compilers and tools, a good option to use by default is `-fast` or `-fastsse`.

These aggregate options incorporate a generally optimal set of flags for targets that support SSE capability. These options incorporate optimization options to enable use of vector streaming SIMD instructions for 64-bit targets. They enable vectorization with SSE instructions, cache alignment, and flushz.



The contents of the `-fast` or `-fastsse` options are host-dependent.

The following table shows the typical `-fast` and `-fastsse` options.

Table 1 Typical `-fast` and `-fastsse` Options

Use this option...	To do this...
<code>-O2</code>	Specifies a code optimization level of 2.

Use this option...	To do this...
<code>-Munroll=c:1</code>	Unrolls loops, executing multiple instances of the original loop during each iteration.
<code>-Mnoframe</code>	Indicates to not generate code to set up a stack frame. Note With this option, a stack trace does not work.
<code>-Mlre</code>	Indicates loop-carried redundancy elimination.
<code>-Mpre</code>	Indicates partial redundancy elimination

`-fast` for 64-bit targets and `-fastsse` for both 32- and 64-bit targets also typically include the options shown in the following table:

Table 2 Additional `-fast` and `-fastsse` Options

Use this option...	To do this...
<code>-Mvect=sse</code>	Generates SSE instructions.
<code>-Mscalarsse</code>	Generates scalar SSE code with xmm registers; implies <code>-Mflushz</code> .
<code>-Mcache_align</code>	Aligns long objects on cache-line boundaries. Note On 32-bit systems, if one file is compiled with the <code>-Mcache_align</code> option, then all files should be compiled with it. This is not necessary on 64-bit systems.
<code>-Mflushz</code>	Sets SSE to flush-to-zero mode.
<code>-M[no]vect</code>	Controls automatic vector pipelining.



For best performance on processors that support SSE instructions, use the PGFORTRAN compiler, even for FORTRAN 77 code, and the `-fastsse` option.

In addition to `-fast` and `-fastsse`, the `-Mipa=fast` option for interprocedural analysis and optimization can improve performance. You may also be able to obtain further performance improvements by experimenting with the individual `-Mpgflag` options that are described in the PGI Compiler Reference Manual, such as `-Mvect`, `-Munroll`, `-Minline`, `-Mconcur`, `-Mpfi`, `-Mpfo`, and so on. However, increased speeds using these options are typically application and system dependent. It is important to time your application carefully when using these options to ensure no performance degradations occur.

Chapter 2.

NEW AND MODIFIED FEATURES

This section provides information about the new or modified features of Release 2014 of the PGI compilers and tools.

2.1. What's New in Release 2014

14.9 Updates and Additions

- ▶ PGI Accelerator Features and Enhancements:
 - ▶ Integrated CUDA 6.5 Toolkit for Linux, Windows (Server 2008 R2 and later) and 64-bit OS X. Refer to [CUDA Fortran Toolkit Issues](#) for more details.
 - ▶ An **acc routine** directive with no parallelism clause (**gang**, **worker** or **vector**) will be treated as if the **seq** clause were present.
 - ▶ When compiling for NVIDIA Tesla targets (using **-ta=tesla** or **-acc** without the **-ta** flag) with the **-Mcuda** flag on the link line, the program will use the default CUDA stream zero for synchronous OpenACC data transfers and kernel launches. This allows OpenACC synchronous data transfers and kernels to interact properly with CUDA synchronous operations. Without the **-Mcuda** option, the program will create a CUDA stream even for synchronous operations, avoiding the serialization associated with CUDA stream zero.
- ▶ A number of problems are corrected in this release. Refer to http://www.pgroup.com/support/release_tprs.htm for a complete and up-to-date table of technical problem reports fixed in recent releases of PGI compilers and tools. This table contains a summary description of each problem as well as the version in which it was fixed.

14.7 Updates and Additions

- ▶ PGI Accelerator Features and Enhancements:
 - ▶ Support for CUDA managed data in CUDA Fortran; refer to the CUDA Fortran Programming Guide and Reference for details.

- ▶ Expanded OpenACC C++ Support
- ▶ Expanded OpenACC 2.0 Features
 - ▶ C global (extern) variables in OpenACC declare directives
 - ▶ Fortran module variables in OpenACC declare directives
 - ▶ Full support for the `atomic` directive
 - ▶ The `wait` clause on OpenACC directives is now supported.
 - ▶ The `async` clause on the `wait` directive is now supported.
 - ▶ When specifying a particular CUDA toolkit version on the command line, if that version is not available in the compiler installation, the compiler will now fail with an error message instead of giving a warning and compiling only for the host.
- ▶ Improved accelerator code generation for nested loops
- ▶ Support for debugging module scope variable in CUDA Fortran
- ▶ New Language Features:
 - ▶ First version to include support for g++ 4.8 compatibility. No versions of PGI prior to 14.7 support GCC 4.8.
 - ▶ New g++ compatibility features in pgc++ including
 - ▶ `__attribute__((used))`
 - ▶ `__attribute__((weak))`
 - ▶ `__attribute__((__constructor__(101)))`
 - ▶ definition of `__LP64__` for 64-bit Linux
 - ▶ definition of `__gnu_linux__`
 - ▶ New F90 pointer optimizations
- ▶ Other Features and Enhancements:
 - ▶ CPU code vectorization enhancements
 - ▶ Support for environment modules in OS X
 - ▶ New Silent Installation option

14.6 Updates and Additions

- ▶ A number of problems are corrected in this release. Refer to http://www.pgroup.com/support/release_tprs.htm for a complete and up-to-date table of technical problem reports fixed in recent releases of PGI compilers and tools. This table contains a summary description of each problem as well as the version in which it was fixed.

14.4 Updates and Additions

- ▶ PGI Accelerator Features and Enhancements:
 - ▶ Expanded OpenACC C++ Support
 - ▶ C++ this pointer support
 - ▶ C++ member functions

- ▶ C++ support for the Routine directive
- ▶ [C++ class member arrays in data clauses](#)
- ▶ Expanded OpenACC 2.0 Features
 - ▶ Loop directive collapse clause on deeply nested loops
 - ▶ Parallel directive firstprivate clause
 - ▶ C structs/[Fortran derived type](#) member arrays in data clauses
 - ▶ Partial support for [Fortran and C/C++ atomic directives](#)
 - ▶ Calling C/C++ CUDA-style atomics from OpenACC
 - ▶ Fortran common block names in OpenACC data clauses
- ▶ GPU-side debugging in OpenACC with Allinea DDT
- ▶ CUDA Fortran support for CUDA 5.5 batched cuBLAS routines
- ▶ Integrated CUDA 6.0 Toolkit
- ▶ New OpenACC tutorial and expanded set of examples
- ▶ PGI Multi-core Features and Enhancements:
 - ▶ Support for new AVX2 instructions available on the latest Haswell CPUs from Intel
 - ▶ Updated Windows assembler
 - ▶ New EDG C++ front-end with C++11 support
- ▶ Other Features and Enhancements
 - ▶ Comprehensive support for environment modules
 - ▶ Prebuilt versions of NetCDF and HDF5

14.2 and 14.3 Updates and Additions

- ▶ A number of problems are corrected in these releases. Refer to http://www.pgroup.com/support/release_tprs.htm for a complete and up-to-date table of technical problem reports fixed in recent releases of PGI compilers and tools. This table contains a summary description of each problem as well as the version in which it was fixed.

14.1 Updates and Additions

- ▶ Updates to PGI OpenACC Fortran/C/C++ compilers, include:
 - ▶ Support for CUDA 5.5 and NVIDIA Kepler K40 GPUs
 - ▶ Support for AMD Radeon GPUs and APUs
 - ▶ Native compilation for NVIDIA and AMD GPUs
 - ▶ Ability within CUDA Fortran to generate dwarf information and debug on the host, device, or both
 - ▶ Additional OpenACC 2.0 features supported, including procedure calls (routine directive), unstructured data lifetimes; create and device_resident clauses for the Declare directive; multidimensional dynamically allocated C/C++ arrays; ability to call CUDA Fortran atomic functions on NVIDIA; and complete run-time API support.

- ▶ PGI Unified Binary for OpenACC programs across NVIDIA and AMD GPUs

For more information, refer to [PGI Accelerator Enhancements](#).

- ▶ Full Fortran 2003 and incremental Fortran 2008 features including long integers, recursive I/O, type statement for intrinsic types, ISO_FORTRAN_ENV and ISO_C_BINDING module updates as well as support for F2008 **contiguous** attribute and keyword.

For more information, refer to [New or Modified Fortran Functionality](#).

- ▶ Extensive updates to libraries:
 - ▶ Updated versions of MPICH, OpenMPI, and MVAPICH pre-built and validated with PGI compilers. For more information, refer to [Using MPI](#).
 - ▶ Pre-packaged open source libraries downloadable from the PGI website including NetCDF and HDF5.
 - ▶ Updated BLAS and LAPACK pre-compiled libraries based on LAPACK 3.4.2.
 - ▶ LAPACK linear algebra math library for shared-memory vector and parallel processors, version 3.4.2, supporting Level 3 BLACS (Basic Linear Algebra Communication Subroutines) for use with PGI compilers. This library is provided in both 64-bit and 32-bit versions for AMD64 or Intel 64 CPU-based installations running Linux, OS X, or Windows.
 - ▶ ScaLAPACK 2.0.2 linear algebra math library for distributed-memory systems for use with MPICH, Open MPI, MVAPICH, and PGI compilers on 64-bit AMD64 or Intel 64 CPU-based installations running Linux and OS X.
 - ▶ Support for the latest Operating Systems including Ubuntu 13.04, Ubuntu 13.10, Fedora 18, Fedora 19, Fedora 20, CentOS 6.4, RHEL 5, RHEL 6, Windows 7, Windows 8, Windows 8.1, OS X Mountain Lion and OSX Mavericks.
 - ▶ GNU compatible C++ improved inlining, Boost and Trilinos correctness as well as OpenACC robustness; full C++11 coming in PGI 14.4.
 - ▶ The `-ta` and `-acc` flags include additional options and functionality. The `-tp` flag functionality is now primarily for processor selection.

For more information, refer to [New or Modified Compiler Options](#).

- ▶ A comprehensive suite of new and updated code examples and tutorials covering Fortran 2003, CUDA Fortran, CUDA-x86, OpenACC, OpenMP parallelization, auto-parallelization, and MPI.
- ▶ These Windows releases are supported in PGI 2014, but will be deprecated in PGI 2015.
 - ▶ Windows XP
 - ▶ Windows Server 2003
 - ▶ Windows Server 2008

2.2. New and Modified Compiler Options

Release 2014 supports a number of new command line options as well as new keyword suboptions for existing command line options.

2.2.1. Required Suboption

The default behavior of the OpenACC compilers has changed in 14.1 from previous releases. The OpenACC compilers now issue a compile-time error if accelerator code generation fails. You can control this behavior with the `required` suboption.

In previous releases, the compiler would issue a warning when accelerator code generation failed. Then it would generate code to run the compute kernel on the host. This previous behavior generates incorrect results if the compute kernels are inside a data region and the host and device memory values are inconsistent.

`-acc=required`, `-ta=tesla:required`, and `-ta=radeon:required` are the defaults.

You can enable the old behavior by using the `norequired` suboption with either of the `-ta` or `-acc` flags.

2.2.2. Accelerator Options



The `-ta=nvidia` option is deprecated in PGI 2014. Users are urged to change their build commands and makefiles to use `-ta=tesla` in place of `-ta=nvidia`.

The `-acc` option enables the recognition of OpenACC directives. In the absence of any explicit `-ta` option, `-acc` implies `-ta=tesla,host`.

-ta Option

The `-ta` option defines the target accelerator and the type of code to generate. This flag is valid for Fortran, C, and C++ on supported platforms.

Syntax

```
-ta=tesla(:tesla_suboptions),radeon(:radeon_suboptions),host
```

There are three major suboptions:

```
tesla(:tesla_suboptions)
radeon(:radeon_suboptions)
host
```

Default

The default is `-ta=tesla,host`.

Select Tesla Accelerator Target

Use the `tesla(:tesla_suboptions)` option to select the Tesla accelerator target and, optionally, to define the type of code to generate.

In the following example, Tesla is the accelerator target architecture and the accelerator generates code for compute capability 3.0:

```
$ pgfortran -ta=tesla:cc30
```

The following table lists and briefly defines the suboptions for the `-ta=tesla` flag.

Table 3 `-ta=tesla` Suboptions

Use this suboption...	To indicate this...
cc10	Generate code for compute capability 1.0.
cc11	Generate code for compute capability 1.1.
cc12	Generate code for compute capability 1.2.
cc13	Generate code for compute capability 1.3.
cc1x	Generate code for the lowest 1.x compute capability possible.
cc1+	Is equivalent to cc1x, cc2x, cc3x.
cc20	Generate code for compute capability 2.0.
cc2x	Generate code for the lowest 2.x compute capability possible.
cc2+	Is equivalent to cc2x, cc3x.
cc30	Generate code for compute capability 3.0.
cc35	Generate code for compute capability 3.5.
cc3x	Generate code for the lowest 3.x compute capability possible.
cc3+	Is equivalent to cc3x.
[no]debug	Enable[disable] debug information generation in device code.
[no]lineinfo	Enable[disable] line information generation in device code.
fastmath	Use routines from the fast math library.
fermi	Is equivalent to cc2x.
fermi+	Is equivalent to cc2+.
[no]flushz	Enable[disable] flush-to-zero mode for floating point computations in the GPU code.
keep	Keep the kernel files.
kepler	Is equivalent to cc3x.

Use this suboption...	To indicate this...
kepler+	Is equivalent to cc3+.
llvm	Generate code using the llvm-based back-end.
maxregcount:n	Specify the maximum number of registers to use on the GPU.
nofma	Do not generate fused multiply-add instructions.
noL1	Prevent the use of L1 hardware data cache to cache global variables.
pin	Set default to pin host memory.
[no]rdc	Generate [do not generate] relocatable device code.
[no]required	Generate [do not generate] a compiler error if accelerator device code cannot be generated.

Select Radeon Accelerator Target

Use the `radeon(:radeon_suboptions)` option to select the Radeon accelerator target and, optionally, to define the type of code to generate.

In the following example, Radeon is the accelerator target architecture and the accelerator generates code for Radeon Cape Verde architecture:

```
$ pgfortran -ta=radeon:capeverde
```

The following table lists and briefly defines the suboptions for the `-ta=radeon` flag.

Table 4 `-ta=radeon` Suboptions

Use this suboption...	To indicate this...
buffercount:n	Set the maximum number of OpenCL buffers in which to allocate data.
capeverde	Generate code for Radeon Cape Verde architecture.
keep	Keep the kernel files.
llvm	Generate code using the llvm-based back-end.
[no]required	Generate [do not generate] a compiler error if accelerator device code cannot be generated.
spectre	Generate code for Radeon Spectre architecture.
tahiti	Generate code for Radeon Tahiti architecture.

Host Option

Use the `host` option to generate code to execute OpenACC regions on the host.

The `-ta=host` flag has no suboptions.

Multiple Targets

Specifying more than one target, such as `-ta=tesla, radeon` generates code for multiple targets. When host is one of the multiple targets, such as `-ta=tesla, host`, the result is generated code that can be run with or without an attached accelerator.

2.2.3. Relocatable Device Code

An `rdc` option is available for the `-ta=tesla` and `-Mcuda` flags that specifies to generate relocatable device code. Starting in PGI 14.1 on Linux and in PGI 14.2 on Windows, the default code generation and linking mode for Tesla-target OpenACC and CUDA Fortran is `rdc`, relocatable device code.

You can disable the default and enable the old behavior and non-relocatable code by specifying any of the following: `-ta=tesla:nordc`, `-Mcuda=nordc`, or by specifying any 1.x compute capability or any Radeon target.

2.2.4. LLVM/SPIR and Native GPU Code Generation

For accelerator code generation, PGI 2014 has two options:

- ▶ In legacy mode, which continues to be the default, PGI generates low-level CUDA C or OpenCL code.
- ▶ Beginning in PGI 14.1, PGI can generate an LLVM-based intermediate representation. To enable this code generation, use `-ta=tesla:llvm` on NVIDIA Tesla hardware or `-ta=radeon:llvm` on AMD Radeon hardware. `-ta=tesla:llvm` implies and requires CUDA 5.5 or higher.

PGI's debugging capability for Tesla uses the LLVM back-end.

2.2.5. DWARF Debugging Formats

PGI 14.4 introduced support for generating dwarf information in GPU code. To enable dwarf generation, just as in host code, you use the `-g` option.

Dwarf generation requires use of the LLVM code generation capabilities. Further, it is possible to generate dwarf information and debug on the host, device, or both. Further, for NVIDIA, the LLVM code generation requires CUDA 5.5.

If you don't want `-g` to apply to both targets, PGI supports the **debug** and **nodebug** suboptions. For example:

- ▶ `-acc -g` implies `-ta=tesla,host -O0 -g` on the host and `-g llvm` on the device with cuda5.5.
- ▶ `-acc -ta=tesla:debug` implies debug on the device; use `llvm` and `cuda5.5`.
- ▶ `-acc -g -ta=tesla:nodebug` implies debug on the host and no `llvm` code generation.

2.2.6. -tp Modifications

The `-tp` switch now truly indicates the target processor. In prior releases a user could use the `-tp` flag to also indicate use of 32-bit or 64-bit code generation. For example, the `-tp shanghai-32` flag was equivalent to the two flags: `-tp shanghai` and `-m32`.

The `-tp` flag interacts with the `-m32` and `-m64` flags to select a target processor and 32-bit or 64-bit code generation. For example, specifying `-tp shanghai -m32` compiles 32-bit code that is optimized for the AMD Shanghai processor, while specifying `-tp shanghai -m64` compiles 64-bit code.

Specifying `-tp shanghai` without a `-m32` or `-m64` flag compiles for a 32-bit target if the PGI 32-bit compilers are on your path, and for a 64-bit target if the PGI 64-bit compilers are on your path.

2.3. New and Modified Fortran Functionality

PVF 2014 contains additional Fortran functionality such as full Fortran 2003 and incremental Fortran 2008 features including long integers, recursive I/O, type statement for intrinsic types, as well as `ISO_FORTRAN_ENV` and `ISO_C_BINDING` module updates and support for F2008 `contiguous` attribute and keyword.

2.3.1. Contiguous Pointers

PGI 2014 supports the `contiguous` attribute as well as the `is_contiguous` intrinsic inquiry function.

contiguous Attribute

Here is an example of a declaration using the `contiguous` keyword:

```
real*4, contiguous, pointer, dimension(:,:) :: arr1_ptr, arr2_ptr, arr3_ptr
```

It is the responsibility of the programmer to assure proper assignment and use of contiguous pointers. Contiguous pointers can result in improved performance, such as this example of using contiguous pointers as the arguments to the `matmul` intrinsic function.

```
arr3_ptr = matmul(arr1_ptr, arr2_ptr)
```

is_contiguous Intrinsic Inquiry Function

The `is_contiguous()` intrinsic function takes a pointer argument and returns a value of type logical. It returns true if the pointer is associated with a contiguous array section, false otherwise.

2.4. New and Modified Tools Functionality

This section provides information about the debugger, PGDBG, and the profiler, PGPROF.

Debug and Profile SGI MPI Programs

In PGI 2014 PGDBG and PGPROF support debugging and profiling of MPI programs built with SGI MPI. To debug an SGI MPI program, use the PGDBG **-sgimpi** option, which has the same syntax as the **-mpi** option.

To profile an SGI MPI program, build it with **-Mprof=func,sgimpi,-Mprof=lines,sgimpi**, or with **-Mprof=time,sgimpi**. You must specify **sgimpi** even if you use **mpicc** or **mpif90** to build your program.

Local and Remote Debugging

PGDBG is licensed software available from The Portland Group. PGDBG supports debugging programs running on local and remote systems. The PGI license keys that enable PGDBG to debug must be located on the same system where the program you want to debug is running.

- ▶ **Local debugging** — If you want to debug a program running on the system where you have launched PGDBG, you are doing local debugging and you need license keys on that local system.
- ▶ **Remote debugging** — If you want to debug a program running on a system other than the one on which PGDBG is launched, then you are doing remote debugging and you need license keys on the remote system. The remote system also needs an installed copy of PGI Workstation, PGI Server, or PGI CDK.

2.5. Using MPI

The PGI compilers provide an option, **-Mmpi**, to make building MPI applications with some MPI distributions more convenient by adding the MPI include and library directories to the compiler's include and library search paths. The compiler determines the location of these directories using various mechanisms.

The following table lists the suboptions supported by **-Mmpi** and briefly describes the required compiling and linking options.

Table 5 MPI Distribution Options

This MPI implementation...	Requires compiling and linking with this option...
MPICH1	Deprecated. -Mmpi=mpich1
MPICH2	Deprecated. -Mmpi=mpich2
MPICH v3	-Mmpi=mpich
MS-MPI	-Mmpi=msmpi
MVAPICH1	Deprecated. -Mmpi=mvapich1
MVAPICH2	Use MVAPICH2 compiler wrappers.
Open MPI	Use Open MPI compiler wrappers.

This MPI implementation...	Requires compiling and linking with this option...
SGI MPI	<code>-Mmpi=sgimpi</code>

For more information on using each of these MPI implementations, refer to *Using MPI* in the PGI Compiler User's Guide.

For distribution of MPI that are not supported by the `-Mmpi` compiler option, use the MPI-distribution-supplied compiler wrappers `mpicc`, `mpic++`, `mpif77`, or `mpif90` to compile and link.

2.6. PGI Accelerator Enhancements

2.6.1. 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 PGI Compiler User's Guide.

2.6.2. CUDA Toolkit Version

The PGI Accelerator x64+accelerator compilers with OpenACC and CUDA Fortran compilers support the CUDA 6.0 toolkit as the default. The compilers and tools also support the CUDA 6.5 Toolkit.

To specify the version of the CUDA Toolkit that is targeted by the compilers, use one of the following properties:

In OpenACC directives

For CUDA Toolkit 6.0: `-ta=tesla:cuda6.0`

For CUDA Toolkit 6.5: `-ta=tesla:cuda6.5`

For CUDA Fortran Construct

For CUDA Toolkit 6.0: `-Mcuda=cuda6.0`

For CUDA Toolkit 6.5: `-Mcuda=cuda6.5`

You may also specify a default version by adding a line to the `siterc` file in the installation `bin/` directory or to a file named `.mypgirc` in your home directory. For example, to specify CUDA Toolkit 6.5, add the following line to one of these files:

```
set DEFCUDAVERSION=6.5;
```

Support for CUDA Toolkit versions 4.2 and earlier has been removed.

2.6.3. 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;
    }
}
```

2.6.4. 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.

```
// 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:

```
// 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 referring 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 incl( int i, elemtype y ){
        data[i] += y;
    }
}
...
#pragma acc parallel loop present(v,x)
    for( int i = 0; i < n; ++i ) v.incl( i, x[i] );
```

As discussed above, the compiler will normally inline **incl**, 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.

2.6.5. 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.

2.6.6. OpenACC Atomic Support

Release 14.7 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);
  }

```

2.6.7. OpenACC declare data directive for global and Fortran module variables

The 14.7 release supports 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 **modx**, 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 **modx** 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], yy[0: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 :: max1 = 100000
  real, dimension(max1) :: 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
```

2.7. C++ Compiler

2.7.1. C++ and OpenACC

This release includes the OpenACC directives for the C++ compilers, (Linux only) `pgc++` and `cpp`. There are limitations to the data that can appear in 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.

- ▶ Any function call in a compute region must be inlined. This includes implicit functions such as for I/O operators, operators on class type, user-defined operators, STL functions, lambda operators, and so on.

2.7.2. C++ Compatibility

PGI 2014 C++ object code is incompatible with prior releases.

All C++ source files and libraries that were built with prior releases must be recompiled to link with PGI 2014 or higher object files.

2.8. New and Modified Runtime Library Routines

PGI 2014 supports new runtime library routines associated with the PGI Accelerator compilers.

For more information, refer to *Using an Accelerator* in the User's Guide.

2.9. Library Interfaces

PGI provides access to a number of libraries that export C interfaces by using Fortran modules. These libraries and functions are described in Chapter 8 of the PGI Compiler Users Guide.

2.10. Environment Modules

On Linux, if you use the Environment Modules package (e.g., the **module load** command), then PGI 2014 includes a script to set up the appropriate module files.

Chapter 3.

DISTRIBUTION AND DEPLOYMENT

Once you have successfully built, debugged and tuned your application, you may want to distribute it to users who need to run it on a variety of systems. This section addresses how to effectively distribute applications built using PGI compilers and tools.

3.1. Application Deployment and Redistributables

Programs built with PGI compilers may depend on runtime library files. These library files must be distributed with such programs to enable them to execute on systems where the PGI compilers are not installed. There are PGI redistributable files for all platforms. On Windows, PGI also supplies Microsoft redistributable files.

3.1.1. PGI Redistributables

The PGI 2014 release includes these directories:

```
$PGI/linux86/14.9/REDIST  
$PGI/linux86-64/14.9/REDIST  
$PGI/win32/14.9/REDIST  
$PGI/win64/14.9/REDIST
```

These directories contain all of the PGI Linux runtime library shared object files, OS X dynamic libraries, or Windows dynamically linked libraries that can be re-distributed by PGI 2014 licensees under the terms of the PGI End-user License Agreement (EULA). For reference, a text-form copy of the PGI EULA is included in the 2014 directory.

3.1.2. Linux Redistributables

The Linux REDIST directories contain the PGI runtime library shared objects for all supported targets. This enables users of the PGI compilers to create packages of executables and PGI runtime libraries that will execute successfully on almost any PGI-supported target system, subject to these requirements:

- ▶ End-users of the executable have properly initialized their environment.
- ▶ Users have set `LD_LIBRARY_PATH` to use the relevant version of the PGI shared objects.

3.1.3. Microsoft Redistributables

The PGI products on Windows include Microsoft Open Tools. The Microsoft Open Tools directory contains a subdirectory named `redist`. PGI 2014 licensees may redistribute the files contained in this directory in accordance with the terms of the PGI End-User License Agreement.

Microsoft supplies installation packages, `vcredist_x86.exe` and `vcredist_x64.exe`, containing these runtime files. These files are available in the `redist` directory.

Chapter 4.

TROUBLESHOOTING TIPS AND KNOWN LIMITATIONS

This section contains information about known limitations, documentation errors, and corrections. Wherever possible, a work-around is provided.

For up-to-date information about the state of the current release, visit the frequently asked questions (FAQ) section on pgroup.com at www.pgroup.com/support/faq.htm

4.1. General Issues

Most issues in this section are related to specific uses of compiler options and suboptions.

- ▶ Object files created with prior releases of PGI compiler are incompatible with object files from PGI 2014 and should be recompiled.
- ▶ Using `-g` option to generate debug information for CUDA Fortran has these limitations:
 - Only linux 64-bit platform supports this feature
 - No debug information is generated for boundaries for Fortran automatic arrays, adjustable dummy arrays, or assumed-shape dummy arrays.
 - No debug information is generated for GPU code after a **!CUF** directive.
- ▶ The `-i8` option can make programs incompatible with the ACML libraries; use of any `INTEGER*8` array size argument can cause failures. Visit developer.amd.com to check for compatible libraries.
- ▶ Using `-Mipa=vestigial` in combination with `-Mipa=libopt` with PGCC, you may encounter unresolved references at link time. This problem is due to the erroneous removal of functions by the `vestigial` sub-option to `-Mipa`. You can work around this problem by listing specific sub-options to `-Mipa`, not including `vestigial`.
- ▶ OpenMP programs compiled using `-mp` and run on multiple processors of a SuSE 9.0 system can run very slowly. These same executables deliver the expected performance and speed-up on similar hardware running SuSE 9.1 and above.

4.2. Platform-specific Issues

4.2.1. Linux

The following are known issues on Linux:

- ▶ Programs that incorporate object files compiled using `-mmodel=medium` cannot be statically linked. This is a limitation of the linux86-64 environment, not a limitation of the PGI compilers and tools.


4.2.2. Apple OS X

The following are known issues on Apple OS X:

- ▶ The PGI 2014 compilers do not support static linking of binaries. For compatibility with future Apple updates, the compilers only support dynamic linking of binaries.
- ▶ Using `-Mprof=func` or `-Mprof=lines` is not supported.

4.2.3. Microsoft Windows

The following are known issues on Windows:

- ▶  Starting January 2015, PGI will no longer include support for Windows XP, Windows Server 2003, or Windows Server 2008.
- ▶ For the Cygwin `emacs` editor to function properly, you must set the environment variable `CYGWIN` to the value `"tty"` before invoking the shell in which `emacs` will run. However, this setting is incompatible with the PGDBG command line interface (`-text`), so you are not able to use `pgdbg -text` in shells using this setting.

The Cygwin team is working to resolve this issue.

- ▶ On Windows, the version of `vi` included in Cygwin can have problems when the `SHELL` variable is defined to something it does not expect. In this case, the following messages appear when `vi` is invoked:

```
E79: Cannot expand wildcards Hit ENTER or type command to continue
```

To workaroud this problem, set `SHELL` to refer to a shell in the cygwin bin directory, e.g. `/bin/bash`.

- ▶ C++ programs on Win64 that are compiled with the option `-tp x64` fail when using PGI Unified Binaries. The `-tp x64` switch is not yet supported on the Windows platform for C++.
- ▶ On Windows, runtime libraries built for debugging (e.g. `msvcrt.d` and `libcmt.d`) are not included with PGI Workstation. When a program is linked with `-g`, for debugging, the standard non-debug versions of both the PGI runtime libraries and the Microsoft runtime libraries are always used. This limitation does not affect debugging of application code.

The following are known issues on Windows and PGDBG:

- ▶ In PGDBG on the Windows platform, Windows times out `stepi/nexti` operations when single stepping over blocked system calls. For more information on the workaround for this issue, refer to the online FAQs at <http://www.pgroup.com/support/tools.htm>.

The following are known issues on Windows and PGPROF:

- ▶ Do not use `-Mprof` with PGI runtime library DLLs. To build an executable for profiling, use the static libraries. When the compiler option `-Bdynamic` is not used, the static libraries are the default.

4.3. PGDBG-related Issues

The following are known issues on PGDBG:

- ▶ Before PGDBG can set a breakpoint in code contained in a shared library, `.so` or `.dll`, the shared library must be loaded.
- ▶ Breakpoints in processes other than the process with rank 0 may be ignored when debugging MPICH-1 applications when the loading of shared libraries to randomized addresses is enabled.
- ▶ Debugging of PGI Unified Binaries, that is, 64-bit programs built with more than one `-tp` option, is not fully supported. The names of some subprograms are modified in the creation, and PGDBG does not translate these names back to the names used in the application source code. For detailed information on how to debug a PGI Unified Binary, refer to <http://www.pgroup.com/support/tools.htm>.

4.4. PGPROF-related Issues

The following are known issues on PGPROF:

- ▶ Accelerator profiling via **pgcollect** is disabled in PGI 2014. PGI Accelerator and OpenACC programs profiled using **pgcollect** do not generate any performance data related to the GPU. This capability is expected to be restored in a future release.

Workaround: Set the environment variable `PGI_ACC_TIME` to '1' for the program when it runs (not when compiling). This setting causes the program to print some performance data to **stdout** on exit.

CUDA Fortran profiling is still available for CUDA Fortran programs.

- ▶ Programs compiled and linked for **gprof**-style performance profiling using `-pg` can result in segmentation faults on systems running version 2.6.4 Linux kernels.
- ▶ Times reported for multi-threaded sample-based profiles, that is, profiling invoked with options `-pg` or `-Mprof=time`, are for the master thread only. To obtain profile data on individual threads, PGI-style instrumentation profiling with `-Mprof={lines | func}` or **pgcollect** must be used.

4.5. CUDA Toolkit Issues

Targeting a CUDA Toolkit Version

- ▶ The CUDA 6.0 Toolkit is set as the default in PGI 14.9. To use the CUDA 6.0 Toolkit, first download the CUDA 6.0 driver from NVIDIA at www.nvidia.com/cuda.

- ▶ You can compile with the CUDA 6.5 Toolkit either by adding the option `-ta=tesla:cuda6.5` to the command line or by adding `set CUDAVERSION=6.5` to the `siterc` file.
- ▶ `pgaccelinfo` prints the driver version as the first line of output. For a 6.5 driver, it prints:

```
CUDA Driver Version 6050
```

CUDA 6.5 Toolkit Limitations

- ▶ Scientific computing libraries including cuBLAS, cuSPARSE, cuFFT and cuRAND are not available for Linux 32-bit.
- ▶ PGI installation packages for the Windows 7/8/2012 operating systems contain both the CUDA 6.0 and 6.5 Toolkits but the packages for the Windows XP/2003/2008 operating systems contain CUDA 6.0 only.
- ▶ The CUDA 6.5 Toolkit is not supported on Windows Server 2012 although it is supported on Windows Server 2012 R2.
- ▶ The CUDA 6.5 Toolkit is not supported on Windows Vista.
- ▶ The CUDA 6.5 Toolkit is not available for OS X 32-bit.

4.6. OpenACC Issues

This section includes known limitations in PGI's support for OpenACC directives.

PGI plans to support these features in a future release, though separate compilation and extern variables for Radeon will be deferred until OpenCL 2.0 is released.

ACC routine directive limitations

- ▶ The `routine` directive has limited support on AMD Radeon. Separate compilation is not supported on Radeon, and selecting `-ta=radeon` disables `rdc` for `-ta=tesla`.
- ▶ The `bind` clause on the `routine` directive is not supported.
- ▶ The `nohost` clause on the `routine` directive is not supported.
- ▶ Reductions in procedures with `acc routine` are not fully supported.
- ▶ Fortran assumed-shape arguments are not yet supported.

Clause Support Limitations

- ▶ The `device_type` clause is not supported on any directive.

4.7. Corrections

A number of problems are corrected in this release. Refer to www.pgroup.com/support/release_tprs.htm for a complete and up-to-date table of technical problem reports, TPRs, fixed in recent releases of the PGI compilers and tools. This table contains a summary description of each problem as well as the version in which it was fixed.

Chapter 5.

CONTACT INFORMATION

You can contact PGI at:

20400 NW Amberwood Drive Suite 100
Beaverton, OR 97006

Or electronically using any of the following means:

Fax: +1-503-682-2637
Sales: sales@pgroup.com
Support: trs@pgroup.com
WWW: <http://www.pgroup.com>

The PGI User Forum is monitored by members of the PGI engineering and support teams as well as other PGI customers. The forum newsgroups may contain answers to commonly asked questions. Log in to the PGI website to access the forum:

<http://www.pgroup.com/userforum/index.php>

Many questions and problems can be resolved by following instructions and the information available at our frequently asked questions (FAQ) site:

<http://www.pgroup.com/support/faq.htm>

All technical support is by e-mail or submissions using an online form at:

<http://www.pgroup.com/support>

Phone support is not currently available.

PGI documentation is available at <http://www.pgroup.com/resources/docs.htm> or in your local copy of the documentation in the release directory `doc/index.htm`.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

PGI Workstation, PGI Server, PGI Accelerator, PGF95, PGF90, PGFORTRAN, and PGI Unified Binary are trademarks; and PGI, PGHPF, PGF77, PGCC, PGC++, PGI Visual Fortran, PVF, PGI CDK, Cluster Development Kit, PGPROF, PGDBG, and The Portland Group are registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2013-2014 NVIDIA Corporation. All rights reserved.

PGI[®]