PGI Visual Fortran® Release Notes

Version 13.10

The Portland Group®

PGI Visual Fortran®

Copyright © 2013 NVIDIA Corporation
All rights reserved.

Printed in the United States of America

First Printing: Release 2013, version 13.2, February 2013
Second Printing: Release 2013, version 13.3, March 2013
Third Printing: Release 2013, version 13.4, April 2013
Fourth Printing: Release 2013, version 13.5, May 2013
Fifth Printing: Release 2013, version 13.6, June 2013
Sixth Printing: Release 2013, version 13.7, July 2013
Seventh Printing: Release 2013, version 13.8, August 2013
Eighth Printing: Release 2013, version 13.9, September 2013
Ninth Printing: Release 2013, version 13.10, October 2013

Technical support: trs@pgroup.com
Sales: sales@pgroup.com
Web: www.pgroup.com

ID: 132821442

Contents

Figures

Tables

Chapter 1. PVF® Release Overview

Welcome to Release 2013 of PGI Visual Fortran®, a set of Fortran compilers and development tools for 32-bit and 64-bit Windows integrated with Microsoft® Visual Studio®.

This document describes the new features of the PVF IDE interface, differences in the PVF 2013 compilers and tools from previous releases, and late-breaking information not included in the standard product documentation.

PGI Visual Fortran (PVF®) is licensed using FLEXnet, the flexible license management system from Flexera Software*. Instructions for obtaining a permanent license are included in your order confirmation. More information on licensing is available in the PVF Installation Guide for this release.

Product Overview

PVF is integrated with several versions of Microsoft Visual Studio. Currently, Visual Studio 2008, 2010 and 2012 are supported. Throughout this document, "PGI Visual Fortran" refers to PVF integrated with any of the three supported versions of Visual Studio. Similarly, "Microsoft Visual Studio" refers to Visual Studio 2008, VS 2010, and VS2012. When it is necessary to distinguish among the products, the document does so.

Single-user node-locked and multi-user network floating license options are available for both products. When a node-locked license is used, one user at a time can use PVF on the single system where it is installed. When a network floating license is used, a system is selected as the server and it controls the licensing, and users from any of the client machines connected to the license server can use PVF. Thus multiple users can simultaneously use PVF, up to the maximum number of users allowed by the license.

PVF provides a complete Fortran development environment fully integrated with Microsoft Visual Studio. It includes a custom Fortran Build Engine that automatically derives build dependencies, Fortran extensions to the Visual Studio editor, a custom PGI Debug Engine integrated with the Visual Studio debugger, PGI Fortran compilers, and PVF-specific property pages to control the configuration of all of these.

Release 2013 of PGI Visual Fortran includes the following components:

- PGFORTRAN OpenMP and auto-parallelizing Fortran 90/95 compiler.
- PGF77 OpenMP and auto-parallelizing FORTRAN 77 compiler.
- PVF Visual Studio integration components.

- AMD Core Math Library (ACML), version 5.24.0 for Windows x64 and version 4.4.0 for 32-bit Windows
- PVF documentation.

If you do not already have Microsoft Visual Studio on your system, be sure to get the PVF installation package that contains the Visual Studio 2012 Shell.

Microsoft Build Tools

PVF on all Windows systems includes the Microsoft Open Tools. On some systems (Windows XP, Windows Server 2003, Windows Server 2008), these files are all the additional tools and libraries required to compile, link, and execute programs on Windows. On other systems (Windows 2008 R2, Windows 7, Windows 8, Windows Server 2012), these files are required in addition to the files Microsoft provides in the Windows 8 SDK.

Terms and Definitions

This Installation Guide contains a number of terms and definitions with which you may or may not be familiar. If you encounter a term in these notes with which you are not familiar, please refer to the online glossary at

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.

Chapter 2. New or Modified Features

This chapter contains the new or modified features of this release of PGI Visual Fortran as compared to prior releases.

What's New in PVF Release 2013

13.10 Updates and Additions

- This release contains a number of changes and additions to the PVF properties:

  Target Accelerators properties:

  - Added NVIDIA: Use L1 Cache to control use of the L1 hardware data cache to cache global variables. (`-ta=nvidia:noL1`prevents use of the L1 hardware data cache to cache global variables.)
  - Replaced NVIDIA: Keep Kernel Binary (`-ta=nvidia:keepbin`) with NVIDIA: Keep Kernel Files (`-ta=nvidia:keep`) which keeps kernel files.
  - Replaced NVIDIA: Keep Kernel Source (`-ta=nvidia:keepgpu`) with NVIDIA: Keep Kernel Files (`-ta=nvidia:keep`) which keeps kernel files.
  - Replaced NVIDIA: Keep Kernel PTX (`-ta=nvidia:keepptx`) with NVIDIA: Keep Kernel Files (`-ta=nvidia:keep`) which keeps kernel files.
  - Removed NVIDIA: Use 24-bit Subscript Mult. (`-ta=nvidia:mul24`)
  - Removed NVIDIA: Synchronous Kernel Launch (`-ta=nvidia:[no]wait`)
  - Removed NVIDIA: Analysis Only (`-ta=nvidia:analysis`)

  Language properties:

  - Added CUDA Fortran Use L1 Cache to control use of the L1 hardware data cache to cache global variables. (`-Mcuda=noL1` prevents use of the L1 hardware data cache to cache global variables.)
- 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 PVF.

13.9 Updates and Additions

- PGI Accelerator x64+GPU native Fortran 2003 compilers and CUDA Fortran support the CUDA 5.0 Toolkit as the default toolkit.

  CUDA 5.0 has deprecated the use of character strings in the symbol API, i.e. functions such as cudaMemcpyToSymbol(). Therefore, CUDA Fortran no longer needs the cudaSymbol derived type, and its definition and use has been removed from the cudafor module. Users should recompile any source code which uses the cudafor module after updating to PGI 13.9 or later.

  PGI compilers and tools also support the CUDA 5.5 Toolkit. For information on specifying the toolkit version, refer to "PGI Accelerator and CUDA Fortran Enhancements," on page 11.
- PGI now enables CUDA Fortran device code to access compute capability 3.x shuffle functions, including `__shfl()`, `__shfl_up()`, `__shfl_down()`, and `__shfl_xor()`. These functions enable access to variables between threads within a warp, referred to as lanes. In CUDA Fortran, lanes use Fortran's 1-based numbering scheme. For more information on these functions, refer to "Shuffle Functions," on page 16.

13.7 Updates and Additions

- OpenACC/CUDA Fortran profiling supports multiple GPUs. In addition, there are improvements to accelerator profiling as a whole. For more information, refer to "OpenACC/CUDA Fortran Profiling," on page 9.
- General performance improvements on AVX vectorized code.
- Support for new CUDA Fortran atomics. For a complete list, refer to "CUDA Fortran Atomic Functions," on page 8.
- PVF 13.7 implements the record format described in *Intel Fortran User's Guide Vol. 1, Building Applications, Variable-Length Records Greater than 2 Gigabytes*. This record format is now the default for the PGI Fortran compilers. To access files written using the old file format, set the environment variable `FORTRANOPT` to `pgi_legacy_large_rec_fmt`.

13.6 Updates and Additions

- Added support for the F2008 *impure* attribute. Fortran 2008 removed the restriction of Fortran 95 that elemental procedures be implicitly pure. Elemental procedures permit writing procedures as many as 16 times, once for each possible rank. In Fortran 2008, elemental procedures must now be explicitly declared with the prefix *impure*.

13.5 Updates and Additions

- Added support for two F2008 features:
  - Finding a unit when opening a file
  - Derived-type-style accesses of the real and imaginary parts of a complex number.
- Made several performance improvements for complex arithmetic, for all languages.

- A number of problems have been 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.

13.4 Updates and Additions

- PGI 13.4 contains support for F2003 non-default derived-type i/o. This feature allows programmers to provide their own formatted and unformatted i/o routines for user-defined or, in certain cases, vendor-supplied derived types. The interfaces for the called subroutines are very specific. For more information on the interfaces refer to a recent Fortran reference manual.

  For a short example of how to use this new feature, refer to "New or Modified Fortran Functionality," on page 8.

- PGI 13.4 contains initial support for F90 pointers in CUDA Fortran device code. In the current implementation, the pointer declaration must be at module scope, in the module which contains the device code. As usual, the host code can use the module and manipulate the pointers. The host code can also pass the pointer as an argument.

  For an example of how to use this new feature, refer to "Using F90 Pointers in CUDA Fortran Device Code," on page 14.

13.3 Updates and Additions

- PVF now supports a new property: `Debugging | Accelerator Profiling`

  Use this property to generate accelerator profiling information at runtime. To do this, set the Accelerator Profiling property to `Yes` which causes the `PGI_ACC_TIME` environment variable to be set to 1 at runtime. The default is `No`.

13.2 Updates and Additions

- PVF 13.2 provides support for F2003 Deferred Type Parameters.
- PVF 13.2 provides ACML version 5.3.0 for 64-bit Windows. For 32-bit Windows, PGI continues to provide ACML version 4.4.0.
- PGI Accelerator x64+GPU native Fortran 2003 compilers and CUDA Fortran support the CUDA 4.2 Toolkit as the default toolkit. PGI compilers and tools also support the CUDA 5.0 Toolkit. For information on specifying the toolkit version, refer to "PGI Accelerator and CUDA Fortran Enhancements," on page 11.
- PVF 13.2 includes expanded properties for new keyword subarguments for existing command line options like `-ta=nvidia` and `-Mcuda`. For more information, refer to "New or Modified Compiler Options," on page 6.
- CUDA Fortran now supports separate compilation and linking of device routines, including device routines in Fortran modules.

  To enable separate compilation and linking, use the CUDA Fortran Generate RDC property to add the command line option `-Mcuda=rdc` to both the compile and the link steps.

- The OpenACC runtime has the concept of device type and device number. For more information, refer to "Device ID and Device Number," on page 13
- In Fortran, there is a new `deviceid` clause for the data, parallel, kernels, update and wait directives. The `deviceid` clause has a single scalar integer argument. For more information, refer to "DEVICEID clause," on page 13
- PGI supports a new command line option, `-Mstack_arrays`, which allows Fortran automatic arrays to be allocated from the stack.

13.1 Updates and Additions

- PGI now supports OpenMP 3.1. Specifically, PGI now supports these features:
  - Fortran, C, and C++ compilers and runtime:
  - omp_in_final runtime routine
  - `OMP_NUM_THREADS` and `OMP_PROC_BIND` environment variables
  - taskyield
  - final tasks and the final clause on task constructs
  - mergeable clause on task constructs
  - atomic read, write, capture, and update clauses
  - min/max reductions in C/C++
  - firstprivate with const data in C/C++, intent(in) in Fortran

New or Modified Compiler Options

PVF 13.10 supports a number of new command line options as well as new keyword subarguments for existing command line options. These changes include:

- `-Mstack_arrays` and `-Mnostack_arrays`

  `-Mstack_arrays` places automatic arrays on the stack while `-Mnostack_arrays` allocates automatic arrays on the heap.
- `-ta=nvidia` has the following new target accelerator arguments:
  - `-ta=nvidia:noL1` prevents the use of L1 hardware data cache to cache global variables.
  - `-ta=nvidia:keep` keeps kernel files.
  - `-ta=nvidia,tesla` is equivalent to `-ta=nvidia,cc1x`
  - `-ta=nvidia,fermi` is equivalent to `-ta=nvidia,cc2x`
  - `-ta=nvidia,kepler` is equivalent to `-ta=nvidia,cc3x`

  PGI continues to support the specific compute capability options (cc10, cc11, etc.) as well as the new cc35 option for compute capability 3.5.

Refer to PVF's Target Accelerators property page when setting compute capability options.

- `-Mcuda` has the following new target accelerator arguments:
  - `-Mcuda=noL1` prevents the use of L1 hardware data cache to cache global variables.
  - `-Mcuda,tesla` is equivalent to `-Mcuda,cclx`
  - `-Mcuda,fermi` is equivalent to `-Mcuda,cc2x`
  - `-Mcuda,kepler` is equivalent to `-Mcuda,cc3x`

  PGI continues to support the specific compute capability options (cc10, cc11, etc.) as well as the new cc35 option for compute capability 3.5.

  Refer to PVF's Language property page when setting compute capability options for CUDA Fortran.
- `-Mcuda` has the new option `rdc` that enables CUDA Fortran separate compilation and linking of device routines, including device routines in Fortran modules. To enable separate compilation and linking, enable PVF's CUDA Fortran Generate RDC property to include the command line option `-Mcuda=rdc` on *both* the compile and the link steps.
- `-Mipa` has the new suboption `-Mipa=reaggregation` that enables IPA-guided structure reaggregation. This automatically attempts to reorder elements in a struct or to split structs into substructs to improve memory locality and cache utilization.
- `-O`

  The optimizations performed at `-O`, `-O1`, `-O2`, `-O3` and `-O4` have changed. These options now have these meanings:

  –O0
  : Level zero specifies no optimization. A basic block is generated for each language statement.

  –O1
  : Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.

  –O
  : When no level is specified, level two global optimizations are performed, including traditional scalar optimizations, induction recognition, and loop invariant motion. No SIMD vectorization is enabled.

  –O2
  : Level two specifies global optimization. This level performs all level-one local optimization as well as level-two global optimization described in –O. In addition, more advanced optimizations such as SIMD code generation, cache alignment, and partial redundancy elimination are enabled.

  –O3
  : Level three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

–04
: Level four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

- `-Mfprelaxed` has the following new suboption: `-Mfprelaxed=intrinsic` that enables use of relaxed precision intrinsics.
- `-tp` has the following new command line target processor options:

  -piledriver
  : generate code that is usable on any Piledriver processor-based system.

  -piledriver-32
  : generate 32-bit code that is usable on any Piledriver processor-based system.

  -piledriver-64
  : generate 64-bit code that is usable on any Piledriver processor-based system.

  Refer to PVF's Target Processors property page to enable compilation for Piledriver or other specific processors.

Unknown options are treated as errors instead of warnings. This feature means it is a compiler error to pass switches that are not known to the compiler; however, you can use the switch `–noswitcherror` to issue warnings instead of errors for unknown switches.

New or Modified Fortran Functionality

PGI 13.10 contains additional fortran functionality.

CUDA Fortran Atomic Functions

PGI 13.10 supports new CUDA Fortran atomics. Table 2.1 indicates the data types supported prior to 13.8 (`prev`) and the additional data types that PGI now supports (`new`) for each atomic function.

Table 2.1. Supported CUDA Fortran Atomic Functions

	integer*4	**integer*8**	**real*4**	**real*8**
`atomicCAS`	`prev`	`new`	`new`	`new`
`atomicADD`	`prev`	`prev`	`prev`	`prev`
`atomicEXCH`	`prev`	`prev`	`prev`	`new`
`atomicSUB`	`prev`	`new`	`new`	`new`
`atomicMAX`	`prev`	`new`	`new`	`new`
`atomicMIN`	`prev`	`new`	`new`	`new`

F2003 Non-default Derived Type I/O

PGI 13.10 supports F2003 non-default derived-type i/o. Using this feature allows programmers to provide their own formatted and unformatted i/o routines for user-defined derived types and for some vendor-supplied derived types.

The following module is one example of a user-defined derived type:

```
module m1
   use iso_c_binding
   interface write(formatted)
     module procedure print_c_ptr
   end interface
   contains
     subroutine print_c_ptr(dtv,unit,iotype,v_list,iostat,iomsg)
       class(c_ptr), intent(in)    :: dtv
       integer, intent(in)         :: unit
       character(*), intent(in)    :: iotype
       integer, intent(in)         :: v_list(:)
       integer, intent(out)        :: iostat
       character(*), intent(inout) :: iomsg
       integer(c_intptr_t)         :: iptrval
       iptrval = transfer(dtv, iptrval)
       if (c_intptr_t.eq.4) then
         write(unit,fmt='("0x",z8.8)') iptrval
       else
         write(unit,fmt='("0x",z16.16)') iptrval
       endif
     end subroutine print_c_ptr
end module m1

program p1
use m1
type(c_ptr) :: p
integer, target :: i, j
p = c_loc(i)
print *,p
end
```

You can compile and run the preceding example using 32 or 64-bit compilers:

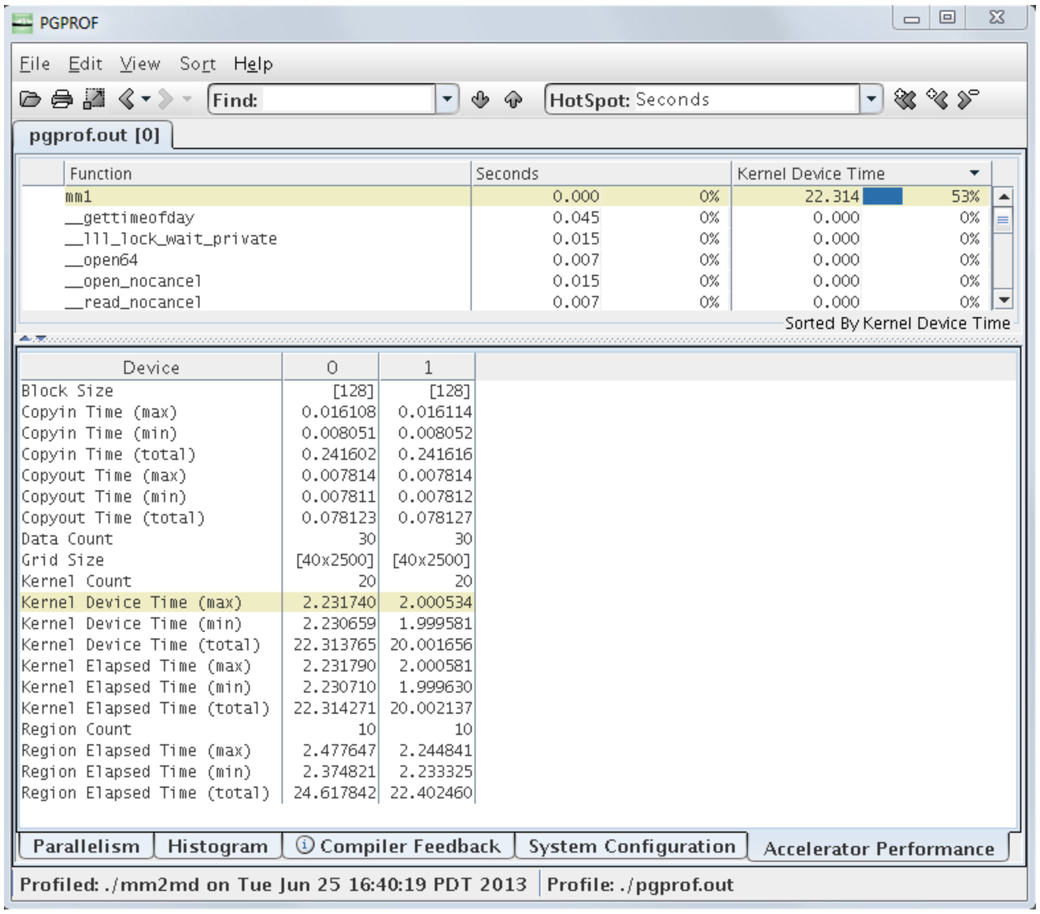
32-bit compilers	64-bit compilers

`% pgf90 -m32 derivedio.f90`
`% ./a.out`
`0x080AA4D4`
	`% pgf90 -m64 derivedio.f90`
	`% ./a.out`
	`0x00007FFF76AD6490`

OpenACC/CUDA Fortran Profiling

In addition to general improvements to accelerator profiling, *PGPROF* includes all-new support for accelerator profiling of multiple devices. As with profiling of single devices, you can view information such as accelerator performance or device configuration. Figure 2.1 shows a view of *PGPROF* that details the accelerator performance on two GPUs for a routine.

Figure 2.1. Accelerator Performance when Profiling Multiple Devices



Function	Seconds		Kernel Device Time	
mm1	0.000	0%	22.314	53%
__gettimeofday	0.045	0%	0.000	0%
__lll_lock_wait_private	0.015	0%	0.000	0%
__open64	0.007	0%	0.000	0%
__open_nocancel	0.015	0%	0.000	0%
__read_nocancel	0.007	0%	0.000	0%

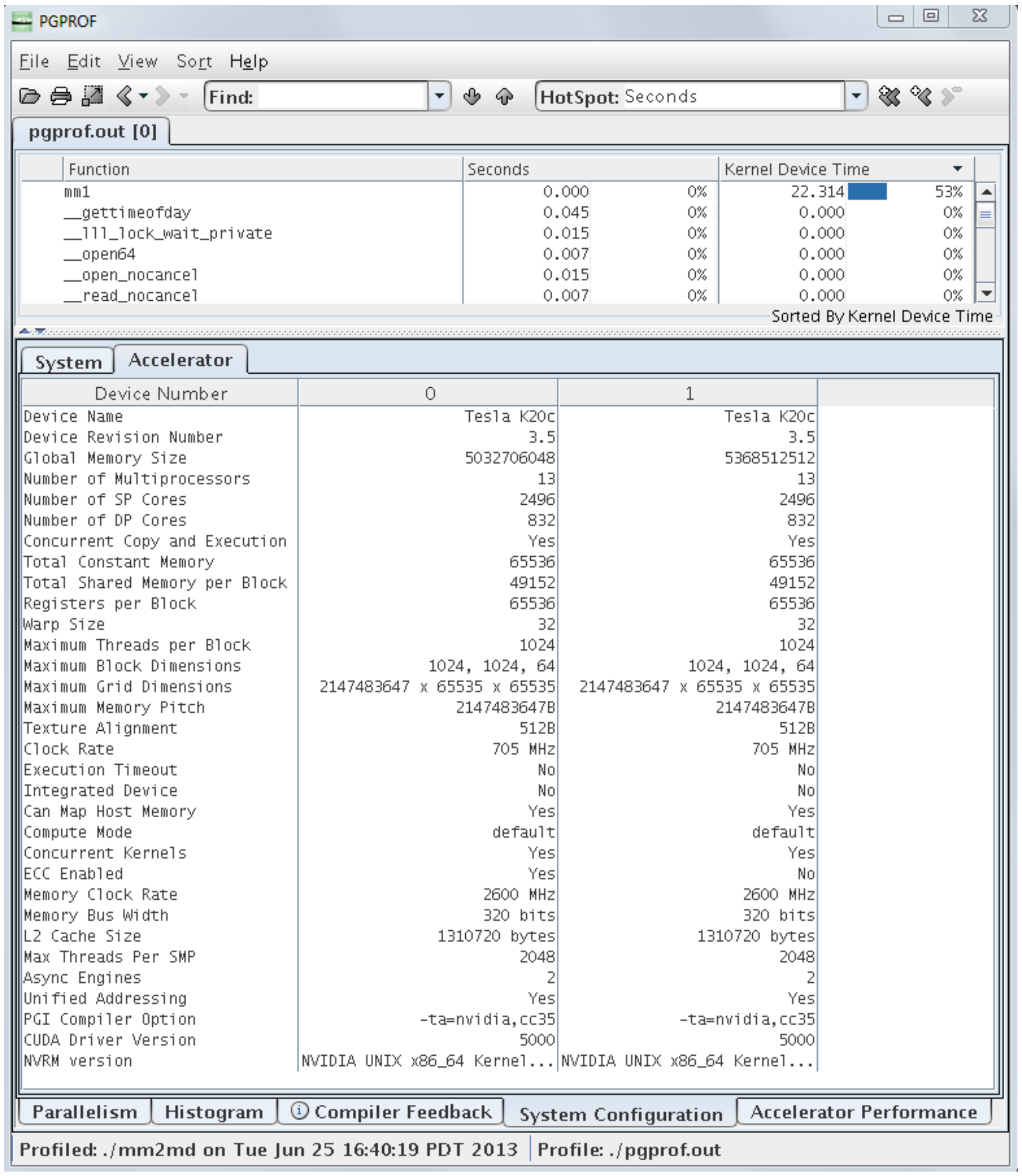
Sorted By Kernel Device Time

Device	0	1
Block Size	[128]	[128]
Copyin Time (max)	0.016108	0.016114
Copyin Time (min)	0.008051	0.008052
Copyin Time (total)	0.241602	0.241616
Copyout Time (max)	0.007814	0.007814
Copyout Time (min)	0.007811	0.007812
Copyout Time (total)	0.078123	0.078127
Data Count	30	30
Grid Size	[40x2500]	[40x2500]
Kernel Count	20	20
Kernel Device Time (max)	2.231740	2.000534
Kernel Device Time (min)	2.230659	1.999581
Kernel Device Time (total)	22.313765	20.001656
Kernel Elapsed Time (max)	2.231790	2.000581
Kernel Elapsed Time (min)	2.230710	1.999630
Kernel Elapsed Time (total)	22.314271	20.002137
Region Count	10	10
Region Elapsed Time (max)	2.477647	2.244841
Region Elapsed Time (min)	2.374821	2.233325
Region Elapsed Time (total)	24.617842	22.402460

Profiled: ./mm2md on Tue Jun 25 16:40:19 PDT 2013 | Profile: ./pgprof.out

Figure 2.2 shows the details about the accelerators on the system where the profile was run.

Figure 2.2. Accelerator Details when Profiling Multiple Devices



Function	Seconds		Kernel Device Time	
mm1	0.000	0%	22.314	53%
__gettimeofday	0.045	0%	0.000	0%
__lll_lock_wait_private	0.015	0%	0.000	0%
__open64	0.007	0%	0.000	0%
__open_nocancel	0.015	0%	0.000	0%
__read_nocancel	0.007	0%	0.000	0%

Sorted By Kernel Device Time

Device Number	0	1
Device Name	Tesla K20c	Tesla K20c
Device Revision Number	3.5	3.5
Global Memory Size	5032706048	5368512512
Number of Multiprocessors	13	13
Number of SP Cores	2496	2496
Number of DP Cores	832	832
Concurrent Copy and Execution	Yes	Yes
Total Constant Memory	65536	65536
Total Shared Memory per Block	49152	49152
Registers per Block	65536	65536
Warp Size	32	32
Maximum Threads per Block	1024	1024
Maximum Block Dimensions	1024, 1024, 64	1024, 1024, 64
Maximum Grid Dimensions	2147483647 x 65535 x 65535	2147483647 x 65535 x 65535
Maximum Memory Pitch	2147483647B	2147483647B
Texture Alignment	512B	512B
Clock Rate	705 MHz	705 MHz
Execution Timeout	No	No
Integrated Device	No	No
Can Map Host Memory	Yes	Yes
Compute Mode	default	default
Concurrent Kernels	Yes	Yes
ECC Enabled	Yes	No
Memory Clock Rate	2600 MHz	2600 MHz
Memory Bus Width	320 bits	320 bits
L2 Cache Size	1310720 bytes	1310720 bytes
Max Threads Per SMP	2048	2048
Async Engines	2	2
Unified Addressing	Yes	Yes
PGI Compiler Option	-ta=nvidia,cc35	-ta=nvidia,cc35
CUDA Driver Version	5000	5000
NVRM version	NVIDIA UNIX x86_64 Kernel...	NVIDIA UNIX x86_64 Kernel...

Profiled: ./mm2md on Tue Jun 25 16:40:19 PDT 2013 | Profile: ./pgprof.out

New or Modified Runtime Library Routines

PGI 2013 supports new runtime library routines associated with the PGI Accelerator compilers. For more information, refer to the "Using an Accelerator" chapter of the PGI Visual Fortran User's Guide.

PGI Accelerator and CUDA Fortran Enhancements

The PGI Accelerator x64+GPU compilers with OpenACC and CUDA Fortran compilers now support the CUDA 5.0 toolkit as the default. The compilers and tools also support the CUDA 5.5 Toolkit. To specify the version of the CUDA Toolkit, use one of the following options:

Since the default toolkit for 13.10 is CUDA 5.0, CUDA Fortran host programs should be recompiled because the PGI 13.3 and higher CUDA Fortran runtime libraries are not compatible with previous CUDA Fortran releases.

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

For PGI Accelerator model:

Use the property: `Fortran | Target Accelerators | NVIDIA: CUDA Toolkit`

When Target NVIDIA Accelerator is set to `Yes`, you can specify the version of the NVIDIA CUDA Toolkit targeted by the compilers.

`Default`: The compiler selects the default CUDA Toolkit version, which is 5.0 for this release.

`5.5`: Specifies use of toolkit version 5.5.
`5.0`: Specifies use of toolkit version 5.0. This is the default.
`4.2`: Specifies use of toolkit version 4.2.
`4.1`: Specifies use of toolkit version 4.1.
`4.0`: Specifies use of toolkit version 4.0.
`3.2`: Specifies use of toolkit version 3.2.
`3.1`: Specifies use of toolkit version 3.1.
`3.0`: Specifies use of toolkit version 3.0.
`2.3`: Specifies use of toolkit version 2.3.

Selecting one of these properties is equivalent to adding the associated switch to the PVF compilation and link lines:

```
–ta=nvidia[:cuda3.2 | cuda4.0 | cuda4.1 | cuda4.2 | cuda5.0 | cuda5.5]
```

For CUDA Fortran:

Use the property: `Fortran | Language | CUDA Fortran Toolkit`

When Enable CUDA Fortran is set to `Yes`, you can specify the version of the CUDA Toolkit targeted by the compilers.

`Default`: The compiler selects the default CUDA Toolkit version, which for 13.10 is version 5.0

`5.5`: Specifies use of toolkit version 5.5.
`5.0`: Specifies use of toolkit version 5.0. This is the default.
`4.2`: Specifies use of toolkit version 4.2.
`4.1`: Specifies use of toolkit version 4.1.
`4.0`: Specifies use of toolkit version 4.0.
`3.2`: Specifies use of toolkit version 3.2.
`3.1`: Specifies use of toolkit version 3.1.
`3.0`: Specifies use of toolkit version 3.0.
`2.3`: Specifies use of toolkit version 2.3.

Selecting one of these properties is equivalent to adding the associated switch to the PVF compilation and link lines:

```
–Mcuda[=cuda3.2 | cuda4.0 | cuda4.1 | cuda4.2 | cuda5.0 | cuda5.5]
```

Device ID and Device Number

The OpenACC runtime has the concept of device type and device number. The supported device types for this release are NVIDIA GPUs (`acc_device_nvidia`) and X86 host (`acc_device_host`). For NVIDIA GPUs, the device number is assigned by the CUDA driver API; the first device is number zero, and other devices have positive numbers.

Device ID

This release supports the concept of *device ID*, which is set by the OpenACC runtime. The available devices are enumerated and given a nonnegative integer device ID, starting at one; the host itself is given the highest device ID value.

The number of available devices can be determined by calling the `acc_get_num_devices` API routine. Since the host is always available as a device, this routine always returns a nonnegative number.

At any point in the program, the runtime keeps track of the current device to be used for the next accelerator construct or directive. The current device can be changed for the host thread by calling the `acc_set_device_type` or `acc_set_device_num` API routines. The current device type and number can be determined by calling the `acc_get_device_type` and `acc_get_device_num` routines.

New, simpler API routines are now supported to allow setting or querying the current device using the device ID.

- `acc_set_deviceid` takes a single integer argument, and sets the current device to the device that corresponds to that device ID.
- `acc_get_deviceid` returns the device ID for the current device.

  A value of zero for a device ID argument corresponds to usingthe current device for this host thread.

If you build your program with the `-ta=nvidia` command line option, only NVIDIA GPU code is generated for the accelerator regions and constructs. At runtime, `acc_get_num_devices` still returns the count of all available devices, including the host, even though your program was not built to run those regions on the host. If your program changes the current device to the host, you get a runtime error when you try to execute a compute region that was only compiled for the NVIDIA GPU.

DEVICEID clause

In Fortran, there is a new `deviceid` clause for the data, parallel, kernels, update and wait directives. The `deviceid` clause has a single scalar integer argument.

- If the `deviceid` clause argument is zero, the current device for this host thread is used.
- If the `deviceid` clause argument is nonzero, the current device for this host thread is changed to the corresponding device for the duration of the region created by this construct.

PGI Accelerator Runtime Routines

PGI_ACC_TIME

When timing the accelerator constructs by setting `PGI_ACC_TIME` to 1, you must run the program with the `LD_LIBRARY_PATH` environment variable set to include the `$PGI/win64/13.10/lib or $PGI/win32/13.10/lib` directory (as appropriate). This release dynamically loads a shared object to implement the profiling feature, and the path to the library must be available.

For a complete description of the PGI Accelerator model runtime routines available in version 13.10, refer to Chapter 4, "PGI Accelerator Compilers Reference" of the *PGI Compiler Reference Manual*.

Memory Management in CUDA

A new memory management routine, `cudaMemGetInfo`, returns the amount of free and total memory available (in bytes) for allocation on the device.

The syntax for `cudaMemGetInfo` is:

```
integer function cudaMemGetInfo( free, total )
     integer(kind=cuda_count_kind) :: free, total
```

Declaring Interfaces to CUDA Device Built-in Routines

A Fortran module is available to declare interfaces to many of the CUDA device built-in routines.

To access this module, add this line to your Fortran program:

```
use cudadevice
```

You can also use these routines in CUDA Fortran global and device subprograms, in CUF kernels, and in PGI Accelerator compute regions in Fortran. Further, the PGI compilers come with implementations of these routines for host code, though these implementations are not specifically optimized for the host.

For a complete list of the CUDA built-in routines that are available, refer to the *PGI CUDA Fortran Programming and Reference*.

Using F90 Pointers in CUDA Fortran Device Code

PGI 13.10 contains support for F90 pointers in CUDA Fortran device code. In the current implementation, the pointer declaration must be at module scope, in the module which contains the device code. The host code can use the module and manipulate the pointers. The host code can also pass the pointer as an argument.

The following example shows how to use F90 pointers in CUDA Fortran as supported by the current release:

```
! Device pointer in module, and passed as argument
module devptr
! pointer declarations must be in the module in which they are used
   real, device, pointer, dimension(:) :: mod_dev_ptr
   real, device, pointer, dimension(:) :: arg_dev_ptr
   real, device, target,  dimension(4) :: mod_dev_arr
   real, device, dimension(4) :: mod_res_arr
```

```
contains
   attributes(global) subroutine test(arg_ptr)
     real, device, pointer, dimension(:) :: arg_ptr
     if (associated(arg_ptr)) then
       mod_res_arr = arg_ptr
     else
       mod_res_arr = mod_dev_ptr
     end if
   end subroutine test
end module devptr

program test
use devptr
real, device, target, dimension(4) :: a_dev
real  result(20)

a_dev = (/ 1.0, 2.0, 3.0, 4.0 /)

! Pointer assignment to device array declared on host,
! passed as argument
arg_dev_ptr => a_dev
call test<<<1,1>>>(arg_dev_ptr)
result(1:4) = mod_res_arr

!$cuf kernel do <<<*,*>>>
do i = 1, 4
   mod_dev_arr(i) = a_dev(i) + 4.0
   a_dev(i)       = a_dev(i) + 8.0
end do

! Pointer assignment to module array, argument nullified
mod_dev_ptr => mod_dev_arr
arg_dev_ptr => null()
call test<<<1,1>>>(arg_dev_ptr)
result(5:8) = mod_res_arr

! Pointer assignment to updated device array, now asssociated
arg_dev_ptr => a_dev
call test<<<1,1>>>(arg_dev_ptr)
result(9:12) = mod_res_arr

!$cuf kernel do <<<*,*>>>
do i = 1, 4
   mod_dev_arr(i) = 25.0 - mod_dev_arr(i)
   a_dev(i)       = 25.0 - a_dev(i)
end do

! Non-contiguous pointer assignment to updated device array
arg_dev_ptr => a_dev(4:1:-1)
call test<<<1,1>>>(arg_dev_ptr)
result(13:16) = mod_res_arr

! Non-contiguous pointer assignment to updated module array
nullify(arg_dev_ptr)
mod_dev_ptr => mod_dev_arr(4:1:-1)
call test<<<1,1>>>(arg_dev_ptr)
result(17:20) = mod_res_arr

print *,result
end
```

You can compile and run the preceding example as follows:

```
% pgf90 devptr.cuf
% ./a.out
    1.000000        2.000000        3.000000        4.000000
    5.000000        6.000000        7.000000        8.000000
    9.000000        10.00000        11.00000        12.00000
    13.00000        14.00000        15.00000        16.00000
    17.00000        18.00000        19.00000        20.00000
```

Shuffle Functions

PGI 13.10 enables CUDA Fortran device code to access compute capability 3.x shuffle functions. These functions enable access to variables between threads within a warp, referred to as *lanes*. In CUDA Fortran, lanes use Fortran's 1-based numbering scheme.

__shfl()

`__shfl()` returns the value of `var` held by the thread whose ID is given by `srcLane`. If the `srcLane` is outside the range of `1:width`, then the thread's own value of `var` is returned. The `width` argument is optional in all shuffle functions and has a default value of 32, the current warp size.

```
    integer(4) function __shfl(var, srcLane, width)
       integer(4) var, srcLane
       integer(4), optional :: width
```

```
    real(4) function __shfl(var, srcLane, width)
       real(4) :: var
       integer(4) :: srcLane
       integer(4), optional :: width
```

```
    real(8) function __shfl(var, srcLane, width)
       real(8) :: var
       integer(4) :: srcLane
       integer(4), optional :: width
```

__shfl_up()

`__shfl_up()` calculates a source lane ID by subtracting delta from the caller's thread ID. The value of `var` held by the resulting thread ID is returned; in effect, `var` is shifted up the warp by `delta` lanes.

The source lane index will not wrap around the value of `width`, so the lower `delta` lanes are unchanged.

```
    integer(4) function __shfl_up(var, delta, width)
       integer(4) var, delta
       integer(4), optional :: width
```

```
    real(4) function __shfl_up(var, delta, width)
       real(4) :: var
       integer(4) :: delta
       integer(4), optional :: width
```

```
    real(8) function __shfl_up(var, delta, width)
       real(8) :: var
       integer(4) :: delta
       integer(4), optional :: width
```

__shfl_down()

`__shfl_down()` calculates a source lane ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` lanes.

The ID number of the source lane will not wrap around the value of width, so the upper `delta` lanes remain unchanged.

```
    integer(4) function __shfl_down(var, delta, width)
       integer(4) var, delta
       integer(4), optional :: width
```

```
    real(4) function __shfl_down(var, delta, width)
       real(4) :: var
       integer(4) :: delta
       integer(4), optional :: width
```

```
    real(8) function __shfl_down(var, delta, width)
       real(8) :: var
       integer(4) :: delta
       integer(4), optional :: width
```

__shfl_xor()

`__shfl_xor()` uses ID-1 to calculate the source lane ID by performing a bitwise XOR of the caller's lane ID with the `laneMask`. The value of `var` held by the resulting lane ID is returned. If the resulting lane ID falls outside the range permitted by `width`, the thread's own value of `var` is returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.

```
    integer(4) function __shfl_xor(var, laneMask, width)
       integer(4) var, laneMask
       integer(4), optional :: width
```

```
    real(4) function __shfl_xor(var, laneMask, width)
       real(4) :: var
       integer(4) :: laneMask
       integer(4), optional :: width
```

```
    real(8) function __shfl_xor(var, laneMask, width)
       real(8) :: var
       integer(4) :: laneMask
       integer(4), optional :: width
```

Here is an example using `__shfl_xor()` to compute the sum of each thread's variable contribution within a warp:

```
        j = . . .
        k = __shfl_xor(j,1);  j = j + k
        k = __shfl_xor(j,2);  j = j + k
        k = __shfl_xor(j,4);  j = j + k
        k = __shfl_xor(j,8);  j = j + k
        k = __shfl_xor(j,16); j = j + k
```

Chapter 3. Selecting an Alternate Compiler

Each release of PGI Visual Fortran contains two components - the newest release of PVF and the newest release of the PGI compilers and tools that PVF targets.

When PVF is installed onto a system that contains a previous version of PVF, the previous version of PVF is replaced. The previous version of the PGI compilers and tools, however, remains installed side-by-side with the new version of the PGI compilers and tools. By default, the new version of PVF will use the new version of the compilers and tools. Previous versions of the compilers and tools may be uninstalled using Control Panel | Add or Remove Programs.

There are two ways to use previous versions of the compilers:

- Use a different compiler release for a single project.
- Use a different compiler release for all projects.

The method to use depends on the situation.

For a Single Project

To use a different compiler release for a single project, you use the compiler flag -V<ver> to target the compiler with version <ver>. This method is the recommended way to target a different compiler release.

For example, -V10.1 causes the compiler driver to invoke the 10.1 version of the PGI compilers if these are installed.

To use this option within a PVF project, add it to the Additional options section of the Fortran | Command Line and Linker | Command Line property pages.

For All Projects

You can use a different compiler release for all projects.

The Tools | Options dialog within PVF contains entries that can be changed to use a previous version of the PGI compilers. Under Projects and Solutions | PVF Directories, there are entries for Executable Directories, Include and Module Directories, and Library Directories.

- For the x64 platform, each of these entries includes a line containing `$(PGIToolsDir)`. To change the compilers used for the x64 platform, change each of the lines containing `$(PGIToolsDir)` to contain the path to the desired bin, include, and lib directories.
- For the 32-bit Windows platform, these entries include a line containing `$(PGIToolsDir)` on 32-bit Windows systems or `$(PGIToolsDir32)` on 64-bit Windows systems. To change the compilers used for the 32-bit Windows platform, change each of the lines containing `$(PGIToolsDir)` or `$(PGIToolsDir32)` to contain the path to the desired bin, include, and lib directories.

Warning

The debug engine in PVF 2013 is not compatible with previous releases. If you use Tools | Options to target a release prior to 2013, you cannot use PVF to debug. Instead, use the `-V` method described earlier in this chapter to select an alternate compiler.

Chapter 4. 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 chapter addresses how to effectively distribute applications built using PGI compilers and tools.

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.

PGI Redistributables

PGI Visual Fortran includes redistributable directories which contain all of the PGI dynamically linked libraries that can be re-distributed by PVF 2013 licensees under the terms of the PGI End-User License Agreement (EULA). For reference, a copy of the PGI EULA in PDF form is included in the release.

The following paths for the redistributable directories assume 'C:' is the system drive.

- On a 32-bit Windows system, the redistributable directory is:

  ```
  C:\Program Files\PGI\win32\13.10\REDIST
  ```

- On a 64-bit Windows system, there are two redistributable directories:

  ```
  C:\Program Files\PGI\win64\13.10\REDIST
  C:\Program Files (x86)\PGI\win32\13.10\REDIST
  ```

The redistributable directories contain the PGI runtime library DLLs for all supported targets. This enables users of the PGI compilers to create packages of executables and PGI runtime libraries that execute successfully on almost any PGI-supported target system, subject to the requirement that end-users of the executable have properly initialized their environment to use the relevant version of the PGI DLLs.

Microsoft Redistributables

PGI Visual Fortran includes Microsoft Open Tools, the essential tools and libraries required to compile, link, and execute programs on Windows. PVF 2013 includes the latest version, version 10, of the Microsoft Open Tools.

The Microsoft Open Tools directory contains a subdirectory named REDIST. PGI 2013 licensees may redistribute the files contained in this directory in accordance with the terms of the associated license agreements.

Note

On Windows, runtime libraries built for debugging (e.g. `msvcrtd` and `libcmtd`) are not included with PGI Visual Fortran. 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.

Chapter 5. Troubleshooting Tips and Known Limitations

This chapter contains information about known limitations, documentation errors, and corrections that have occurred to PVF 2013. Whenever possible, a workaround 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/index.htm

Use MPI in PVF Limitations

- The multi-process debug style known as "Run One At a Time" is not supported in this release.

PVF IDE Limitations

The issues in this section are related to IDE limitations.

- Integration with source code revision control systems is not supported.
- When moving a project from one drive to another, all .d files for the project should be deleted and the whole project should be rebuilt. When moving a solution from one system to another, also delete the solution's Visual Studio Solution User Options file (.suo).
- The Resources property pages are limited. Use the Resources | Command Line property page to pass arguments to the resource compiler. Resource compiler output must be placed in the intermediate directory for build dependency checking to work properly on resource files.
- There are several properties that take paths or pathnames as values. In general, these may not work as expected if they are set to the project directory $(ProjectDir) or if they are empty, unless empty is the default. Specifically:

  General | *Output Directory* should not be empty or set to $(ProjectDir).
  General | *Intermediate Directory* should not be empty or set to $(ProjectDir).
  Fortran | *Output* | *Object File Name* should not be empty or set to $(ProjectDir).
  Fortran | *Output* | *Module Path* should not be empty or set to include $(ProjectDir).

- Dragging and dropping files in the Solution Explorer that are currently open in the Editor may result in a file becoming "orphaned." Close files before attempting to drag-and-drop them.

PVF Debugging Limitations

The following limitations apply to PVF debugging:

- Debugging of unified binaries is not fully supported. The names of some subprograms are modified in the creation of the unified binary, and the PVF debug engine does not translate these names back to the names used in the application source code. For more information on debugging a unified binary, see www.pgroup.com/support/tools.htm.
- In some situations, using the Watch window may be unreliable for local variables. Calling a function or subroutine from within the scope of the watched local variable may cause missed events and/or false positive events. Local variables may be watched reliably if program scope does not leave the scope of the watched variable.
- Rolling over Fortran arrays during a debug session is not supported when Visual Studio is in Hex mode. This limitation also affects Watch and Quick Watch windows.

  Workaround: deselect Hex mode when rolling over arrays.

PGI Compiler Limitations

The frequently asked questions (FAQ) section on the pgroup.com web page at www.pgroup.com/support/index.htm provides more up to date information about the state of the current release.

- Take extra care when using `-Mprof` with PVF runtime library DLLs. To build an executable for profiling, use of the static libraries is recommended. The static libraries are used by default in the absence of `-Bdynamic`.
- Using `–Mpfi` and `–mp` together is not supported. The `–Mpfi` flag disables `–mp` at compile time, which can cause runtime errors in programs that depend on interpretation of OpenMP directives or pragmas. Programs that do not depend on OpenMP processing for correctness can still use profile feedback. Using the `–Mpfo` flag does not disable OpenMP processing.
- The `–i8` option can make programs incompatible with the ACML library; use of any INTEGER*8 array size argument can cause failures with these libraries. Visit developer.amd.com to check for compatible ACML libraries.
- ACML 5.1.0 is built using the `–fastsse` compile/link option, which includes `–Mcache_align`. When linking with ACML on 32-bit Windows, all program units must be compiled with `–Mcache_align`, or an aggregate option such as `–fastsse`, which incorporates `–Mcache_align`. This process is not an issue on 64-bit targets where the stack is 16-byte aligned by default. You can use the lower-performance, but fully portable, `blas` and `lapack` libraries on CPUs that do not support SSE instructions.

Corrections

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

Chapter 6. Contact Information

You can contact The Portland Group at:

The Portland Group
Two Centerpointe Drive, Suite 320
Lake Oswego, OR 97035 USA

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:

www.pgroup.com/userforum/index.php

Or contact us electronically using any of the following means:

Fax	+1-503-682-2637
Sales	sales@pgroup.com
Support	trs@pgroup.com
WWW	www.pgroup.com

All technical support is by email or submissions using an online form at www.pgroup.com/support. Phone support is not currently available.

Many questions and problems can be resolved at our frequently asked questions (FAQ) site at www.pgroup.com/support/faq.htm.

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