

PGI[®] COMPILERS & TOOLS

RELEASE NOTES FOR X86 CPUS AND TESLA GPUS

Version 2018



TABLE OF CONTENTS

Chapter 1. What's New in PGI 2018.....	1
1.1. What's New in 18.7.....	1
1.2. What's New in 18.5.....	4
1.3. What's New in 18.4.....	4
1.4. What's New in 18.3.....	4
1.5. What's New in 18.1.....	4
1.6. OpenACC.....	8
1.6.1. OpenACC Error Handling.....	9
1.6.2. Performance Impact of Fortran 2003 Allocatables.....	12
1.7. OpenMP.....	12
1.8. PCAST Overview.....	13
1.9. Deep Copy Overview.....	16
1.10. C++ Compiler.....	19
1.10.1. C++17.....	19
1.10.2. C++ and OpenACC.....	20
1.10.3. C++ Compatibility.....	21
1.11. Runtime Library Routines.....	21
1.12. Library Interfaces.....	21
1.13. Environment Modules.....	21
1.14. LLVM Code Generator.....	22
Chapter 2. Release Overview.....	23
2.1. Licensing.....	23
2.1.1. Licensing Terminology.....	23
2.1.2. Bundled License Key.....	24
2.1.3. Node-locked and Network Floating Licenses.....	24
2.2. Release Components.....	24
2.2.1. Additional Components.....	25
2.2.2. MPI Support.....	25
2.3. Terms and Definitions.....	25
2.4. Supported Platforms.....	25
2.5. Supported Operating System Updates.....	26
2.5.1. Linux.....	26
2.5.2. Apple macOS.....	26
2.5.3. Microsoft Windows.....	26
2.6. CUDA Toolkit Versions.....	27
2.7. Compute Capability.....	29
2.8. Precompiled Open-Source Packages.....	29
2.9. Getting Started.....	31
Chapter 3. Distribution and Deployment.....	33
3.1. Application Deployment and Redistributables.....	33

3.1.1. PGI Redistributables.....	33
3.1.2. Linux Redistributables.....	33
3.1.3. Microsoft Redistributables.....	34
Chapter 4. Troubleshooting Tips and Known Limitations.....	35
4.1. Platform-specific Issues.....	35
4.1.1. Linux.....	35
4.1.2. Apple macOS.....	35
4.1.3. Microsoft Windows.....	35
4.2. Issues Related to Debugging.....	36
4.3. Profiler-related Issues.....	36
4.4. OpenACC Issues.....	36
Chapter 5. Contact Information.....	38

LIST OF TABLES

Table 1	Third-Party Software Security Updates for PGI version 18.7	8
Table 2	GCC Version Compatibility with -ta=tesla:nollvm	21
Table 3	Typical -fast Options	31
Table 4	Additional -fast Options	31

Chapter 1.

WHAT'S NEW IN PGI 2018



Important The PGI 2018 Release includes updated FlexNet license management software to address a [security vulnerability](#). Users of any previous PGI release must update their FlexNet license daemons to enable PGI 18.1 and subsequent releases. See Third-Party Software Security Updates below and our [FlexNet Update FAQ](#) for more information.

Welcome to Release 2018 of the PGI compilers and tools!

If you read only one thing about this PGI release, make it this chapter. It covers all the new, changed, deprecated, or removed features in PGI products released this year. It is written with you, the user, in mind.

Every PGI release contains user-requested fixes and updates. We keep a complete list of these fixed [Technical Problem Reports](#) online for your reference.

1.1. What's New in 18.7

All Compilers

The LLVM-based code generator for Linux/x86-64 and Linux/OpenPOWER platforms is now based on LLVM 6.0. On Linux/x86-64 targets where it remains optional (using the `-Mllvm` compiler flag), performance of generated executables using the LLVM-based code generator average 15% faster than the default PGI code generator on several important benchmarks. The LLVM-based code generator will become default on all x86-64 targets in a future PGI release.

The PGI compilers are now interoperable with GNU versions up to and including GCC 8.1. This includes interoperability between the PGI C++ compiler and g++ 8.1, and the ability for all of the PGI compilers to interoperate with the GCC 8.1 toolchain components, header files and libraries.

Fortran

Implemented Fortran 2008 SUBMODULE support. A submodule is a program unit that extends a module or another submodule.

The default behavior for assignments to allocatable variables has been changed to match Fortran 2003 semantics in both host and GPU device code generation. This change may affect performance in some cases where a Fortran allocatable array assignment is made within a kernels directive. For more information and a suggested workaround, refer to [Performance Impact of Fortran 2003 Allocatables](#). This change can be reverted to the pre-18.7 behavior of Fortran 1995 semantics on a per-compilation basis by adding the `-Mallocatable=95` option.

When using the LLVM-based code generator, the Fortran compiler now generates LLVM debug metadata for module variables, and the quality of debug metadata is generally improved.

Free-form source lines can now be up to 1000 characters.

All Fortran CHARACTER entities are now represented with a 64-bit integer length.

OpenACC and CUDA Fortran

Added support for an implementation of the draft OpenACC 3.0 true deep copy directives for aggregate data structures in Fortran, C and C++. With true deep copy directives you can specify a subset of members to move between host and device memory within the declaration of an aggregate, including support for named policies that allow distinct sets of members to be copied at different points in a program. For more information, see [Deep Copy Overview](#).

Added support for PGI Compiler Assisted Software Testing (PCAST) features including OpenACC autocompare. The new `-ta=tesla:autocompare` compiler option causes OpenACC compute regions to run redundantly on both the CPU and GPU, and GPU results are compared with those computed on the CPU. PCAST can also be used in the form of new run-time `pgi_compare` or `acc_compare` API calls. In either case, PCAST compares computed results with known correct values and reports errors if the data does not match within some user-specified tolerance. Both forms are useful for pinpointing computational divergences between a host CPU and a GPU, and the API calls can be used more generally to investigate numerical differences in a program compiled for execution on different processor architectures. See [PCAST Overview](#) for more details on using the new PCAST features.

The compilers now set the default CUDA version to match the CUDA Driver installed on the system used for compilation. CUDA 9.1 and CUDA 9.2 toolkits are bundled with the PGI 18.7 release. Older CUDA toolchains including CUDA 8.0 and CUDA 9.0 have been removed from the PGI installation packages to minimize their size, but are still supported using a new co-installation feature. If you do not specify a `cudaX.Y` sub-

option to `-ta=tesla` or `-Mcuda`, you should read more about how this change affects you in [CUDA Toolkit Versions](#).

Changed the default NVIDIA GPU compute capability list. The compilers now construct the default compute capability list to be the set matching that of the GPUs installed on the system on which you are compiling. If you do not specify a compute capability sub-option to `-ta=tesla` or `-Mcuda`, you should read more about how this change affects you in [Compute Capability](#).

Changed the default size for `PGI_ACC_POOL_ALLOC_MINSIZE` to 128B from 16B. This environment variable is used by the accelerator compilers' CUDA Unified Memory pool allocator. A user can revert to pre-18.7 behavior by setting `PGI_ACC_POOL_ALLOC_MINSIZE` to 16B.

Added an example of how to use the OpenACC error handling callback routine to intercept errors triggered during execution on a GPU. Refer to [OpenACC Error Handling](#) for explanation and code samples.

Added support for assert function call within OpenACC accelerator regions on all platforms.

OpenMP

Improved the efficiency of code generated for OpenMP 4.5 combined "distribute parallel" loop constructs that include a collapse clause, resulting in improved performance on a number of applications and benchmarks.

When using the LLVM-based code generator, the Fortran compiler now generates DWARF debug information for OpenMP thread private variables.

Utilities

Changed the output of the tool `pgacclinfo`. The label of the last line of `pgacclinfo`'s output is now "PGI Default Target" whereas prior to the 18.7 release the label "PGI Compiler Option" was used.

Changed the output of the tool `pgcpuid`. The label of the last line of `pgcpuid`'s output is now "default target" whereas prior to the 18.7 release the label "type" was used.

Deprecations and Eliminations

Dropped support for NVIDIA Fermi GPUs; compilation for compute capability 2.x is no longer supported.

PGI 18.7 is the last release for Windows that includes bundled Microsoft toolchain components. Future releases will require users to have the Microsoft toolchain components pre-installed on their systems.

1.2. What's New in 18.5

The PGI 18.5 release contains all features and fixes found in PGI 18.4 and a few key updates for important user-reported problems.

Added full support for CUDA 9.2; use the `cuda9.2` sub-option with the `-ta=tesla` or `-Mcuda` compiler options to compile and link with the integrated CUDA 9.2 toolkit components.

Added support for Xcode 9.3 on macOS.

Improved function offset information in runtime tracebacks in optimized (non-debug) modes.

1.3. What's New in 18.4

The PGI 18.4 release contains all features and fixes found in PGI 18.3 and a few key updates for important user-reported problems.

Added support for CUDA 9.2 if one directs the compiler to a valid installation location of CUDA 9.2 using `CUDA_HOME`.

Added support for internal procedures, assigned procedure pointers and internal procedures passed as actual arguments to procedure dummy arguments.

Added support for intrinsics `SCALE`, `MAXVAL`, `MINVAL`, `MAXLOC` and `MINLOC` in initializers.

1.4. What's New in 18.3

The PGI 18.3 release contains all the new features found in PGI 18.1 and a few key updates for important user-reported problems.

C/C++

Implemented the `__builtin_return_address` and `__builtin_frame_address` functions; available on Linux/x86 with the LLVM code generator compilers.

1.5. What's New in 18.1

Key Features

Added support for Intel Skylake and AMD Zen processors, including support for the AVX-512 instruction set on the latest Intel Xeon processors.

Added full support for OpenACC 2.6.

Enhanced support for OpenMP 4.5 for multicore CPUs, including SIMD directives as tuning hints. OpenMP 4.5 is supported on Linux/x86 with the LLVM code generator (see below).

Added support for the CUDA 9.1 toolkit, including on the latest NVIDIA Volta V100 GPUs.

OpenACC and CUDA Fortran

Changed the default CUDA Toolkit used by the compilers to CUDA Toolkit 8.0.

Changed the default compute capability chosen by the compilers to cc35,cc50,cc60 .

Added support for CUDA Toolkit 9.1.

Added full support for the OpenACC 2.6 specification including:

- ▶ serial construct
- ▶ if and if_present clauses on host_data construct
- ▶ no_create clause on the compute and data constructs
- ▶ attach clause on compute, data, and enter data directives
- ▶ detach clause on exit data directives
- ▶ Fortran optional arguments
- ▶ acc_get_property, acc_attach, and acc_detach routines
- ▶ profiler interface

Added support for asterisk (*) syntax to CUDA Fortran launch configuration. Providing an asterisk as the first execution configuration parameter leaves the compiler free to calculate the number of thread blocks in the launch configuration.

Added two new CUDA Fortran interfaces, `cudaOccupancyMaxActiveBlocksPerMultiprocessor` and `cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags`. These provide hooks into the CUDA Runtime for manually obtaining the maximum number of thread blocks which can be used in grid-synchronous launches, same as provided by the asterisk syntax above.

OpenMP

Changed the default initial value of `OMP_MAX_ACTIVE_LEVELS` from 1 to 16.

Added support for the **taskloop** construct's **firstprivate** and **lastprivate** clauses.

Added support for the OpenMP Performance Tools (OMPT) interface. Available with the LLVM code generator compilers on Linux.

C++

Added support for GNU interoperability through GCC 7.2.

Added partial support for C++17 including constexpr if, fold expressions, structured bindings, and several other C++17 features. See [C++17](#) for a complete list of supported features.

Fortran

Changed how the PGI compiler runtime handles Fortran array descriptor initialization; this change means any program using Fortran 2003 should be recompiled with PGI 18.1.

Libraries

Reorganized the Fortran cuBLAS and cuSolver modules to allow use of the two together in any Fortran program unit. As a result of this reorganization, any codes which use cuBLAS or cuSolver modules must be recompiled to be compatible with this release.

Added a new PGI math library, libpgm. Moved math routines from libpgc, libpgftnrtl, and libpgf90rtl to libpgm. This change should be transparent unless you have been explicitly adding libpgc, libpgftnrtl, or libpgf90rtl to your link line.

Added new fastmath routines for single precision scalar/vector sin/cos/tan for AVX2 and AVX512F processors.

Added support for C99 scalar complex intrinsic functions.

Added support for vector complex intrinsic functions.

Added environment variables to control runtime behavior of intrinsic functions:

MTH_I_ARCH={em64t,sse4,avx,avxfma4,avx2,avx512knl,avx512}

Override the architecture/platform determined at runtime.

MTH_I_STATS=1

Provide basic runtime statistics (number of calls, number of elements, percentage of total) of elemental functions.

MTH_I_STATS=2

Provide detailed call count by element size (single/double-precision scalar, single/double-precision vector size).

MTH_I_FAST={relaxed,precise}

Override compile time selection of fast intrinsics (the default) and replace with either the relaxed or precise versions.

MTH_I_RELAXED={fast,precise}

Override compile time selection of relaxed intrinsics (the default with `-Mfprelaxed=intrinsic`) and replace with either the fast or precise versions.

MTH_I_PRECISE={fast,relaxed}

Override compile time selection of precise intrinsics (the default with `-Kieee`) and replace with either the fast or relaxed versions.

Profiler

Improved the CPU Details View to include the breakdown of time spent per thread.

Added an option to let one select the PC sampling frequency.

Enhanced the NVLink topology to include the NVLink version.

Enhanced profiling data to include correlation ID when exporting in CSV format.

Operating Systems and Processors

Added support for the AMD Zen (EPYC, Ryzen) processor architecture. Use the `-tp=zen` compiler option to target AMD Zen explicitly.

Added support for the Intel Skylake processor architecture. Use the `-tp=skylake` compiler option to target Intel Skylake explicitly.

Added support for the Intel Knights Landing processor architecture. Use the `-tp=knl` compiler option to target Intel Knights Landing explicitly.

LLVM Code Generator

Released a production version of the PGI Linux/x86-64 compilers with an LLVM code generator and OpenMP runtime; these compilers can be found in the PGI installation directory `linux86-64-llvm` installed alongside the default PGI x86-64 compilers in `linux86-64`. See [LLVM Code Generator](#) for more information.

License Management

Updated FlexNet Publisher license management software to v11.14.1.3. This update addresses several issues including:

- ▶ A security vulnerability on Windows. See Third-Party Software Security Updates below and the [FlexNet Update FAQ](#) for more information.
- ▶ Seat-count stability improvements on network floating license servers when borrowing licenses (`lmborrow`) for off-line use. For early return of borrowed seats, users should invoke the new `"-bv"` option for `lmborrow`. See our [license borrowing FAQ](#) for more information.



Important Users with PGI 2017 (17.x) or older need to update their license daemons to support 18.1 or newer. The new license daemons are backward-compatible with older PGI releases.

Deprecations and Eliminations

Dropped support for the following versions of Linux:

- ▶ CentOS 5 through 6.3

- ▶ Fedora 6 through 13
- ▶ openSUSE 11 through 13.1 and openSUSE Leap through 42.2
- ▶ RHEL through 6.3
- ▶ SLES 11
- ▶ Ubuntu 12.04

Dropped support for versions of glibc older than 2.12.

Dropped support for macOS version 10.9 (Mavericks).

Stopped including components from CUDA Toolkit version 7.5 in the PGI packages. CUDA 7.5 can still be targeted if one directs the compiler to a valid installation location of CUDA 7.5 using `CUDA_HOME`.

Deprecated legacy PGI accelerator directives. When the compiler detects a deprecated PGI accelerator directive, it will print a warning. This warning will include the OpenACC directive corresponding to the deprecated directive if one exists. Warnings about deprecated directives can be suppressed using the new `legacy` sub-option to the `-acc` compiler option. The following library routines have been deprecated: `acc_set_device`, `acc_get_device`, and `acc_async_wait`; they have been replaced by `acc_set_device_type`, `acc_get_device_type`, and `acc_wait`, respectively. The following environment variables have been deprecated: `ACC_NOTIFY` and `ACC_DEVICE`; they have been replaced by `PGI_ACC_NOTIFY` and `PGI_ACC_DEVICE_TYPE`, respectively. Support for legacy PGI accelerator directives may be removed in a future release.

Dropped support for CUDA x86. The `-Mcuda x86` compiler option is no longer supported.

Dropped support for CUDA Fortran emulation mode. The `-Mcuda=emu` compiler option is no longer supported.

Third-Party Software Security Updates

Table 1 Third-Party Software Security Updates for PGI version 18.7

CVE ID	Description
CVE-2016-10395	Updated FlexNet Publisher to v11.14.1.3 to address a vulnerability on Windows. We recommend all users update their license daemons –see the FlexNet Update FAQ . For more information, see the Flexera website .

1.6. OpenACC

The following sections contain details about updates to PGI compiler support for OpenACC.

1.6.1. OpenACC Error Handling

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

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

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

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

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



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

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

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

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

#define N 2147483648

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

    MPI_Init(&argc, &argv);

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

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

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

```

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

    // do some more work

    MPI_Finalize();

    return 0;
}

```

We compile the program with:

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

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

```

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

-----
Primary job terminated normally, but 1 process returned
a non-zero exit code.. Per user-direction, the job has been aborted.
-----

-----
mpirun detected that one or more processes exited with non-zero status,
thus causing the job to be terminated.

```

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

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

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

```

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

#define N 2147483648

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

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

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

    MPI_Init(&argc, &argv);

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

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

        acc_set_error_routine(&handle_gpu_errors);

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

    // more work

    MPI_Finalize();

    return 0;
}

```

Again, we compile the program with:

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

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

```
$ mpirun -n 2 ./error_handling_mpi
```

```

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

```

```

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

```

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

1.6.2. Performance Impact of Fortran 2003 Allocatables

In the PGI 18.7 release, use of Fortran 2003 semantics for assignments to allocatables was made the default. This change applies to host and device code alike. Previously, Fortran 1995 semantics were followed by default. The change to Fortran 2003 semantics may affect performance in some cases where the `kernels` directive is used.

When the following Fortran allocatable array assignment is compiled using the Fortran 2003 specification, the compiler cannot generate parallel code for the array assignment; lack of parallelism in this case may negatively impact performance.

```

real, allocatable, dimension(:) :: a, b
allocate(a(100), b(100))
a = 3.14

!$acc kernels
a = b
!$acc end kernels

```

The example code can be modified to use an array section assignment instead; the compiler can parallelize the array section assignment and the lost performance is regained.

```

a(:) = b(:)

```

1.7. OpenMP

OpenMP 3.1

The PGI Fortran, C, and C++ compilers support OpenMP 3.1 on all platforms.

OpenMP 4.5

The PGI Fortran, C, and C++ compilers compile most OpenMP 4.5 programs for parallel execution across all the cores of a multicore CPU or server. **target** regions are implemented with default support for the multicore host as the target, and **parallel** and **distribute** loops are parallelized across all OpenMP threads. This feature is supported on Linux/x86 platforms with the LLVM code generator only.

Current limitations include:

- ▶ The **simd** construct can be used to provide tuning hints; the **simd** construct's **private**, **lastprivate**, **reduction**, and **collapse** clauses are processed and supported.
- ▶ The **declare simd** construct is ignored.
- ▶ The **ordered** construct's **simd** clause is ignored.
- ▶ The **task** construct's **depend** and **priority** clauses are not supported.
- ▶ The loop construct's **linear**, **schedule**, and **ordered(n)** clauses are not supported.
- ▶ The **declare reduction** directive is not supported.

1.8. PCAST Overview

PGI Compiler Assisted Software Testing (PCAST) is a set of functionality to help test for program correctness and determine points of divergence. There are three different ways to invoke PCAST; through `pgi_compare` or `acc_compare` run-time calls, or with the autocompare compiler flag.

Let's look at the different ways you can utilize PCAST with a simple example. The following C program allocates two arrays of some `size` on the heap, copies the data to the GPU, and creates gangs of workers to execute the inner loop. In the next few paragraphs, we'll demonstrate different ways to use PCAST to test for program correctness.

```
int main() {
    int size = 1000;
    int i, t;
    float *a1;
    float *a2;

    a1 = (float*)malloc(sizeof(float)*size);
    a2 = (float*)malloc(sizeof(float)*size);

    for (i = 0; i < size; i++) {
        a1[i] = 1.0f;
        a2[i] = 2.0f;
    }

    #pragma acc data copy(a1[0:size], a2[0:size])
    {
        for (t = 0; t < 5; t++) {
            #pragma acc parallel
            for(i = 0; i < size; i++) {
                a2[i] += a1[i];
            }
        }
    }

    return 0;
}
```

The first, and simplest, way to invoke PCAST is through the use of the autocompare compiler flag. Setting `-ta=tesla:autocompare` in the compiler options is the

only change necessary to invoke the autocompare feature. When compiled with this option, code in OpenACC compute regions will run redundantly on the CPU as well as the GPU. Whenever computed data is copied off the GPU and back into host memory, it is compared against the values computed on the CPU. Hence, any data in a `copy`, `copyout`, or `update host` directive will be compared. Note that the `-ta=tesla:autocompare` implies `-ta=tesla:redundant`.

We can compile our example with:

```
$ pgcc -Minfo=accel -ta=tesla:autocompare -o a.out example.c
```

Before we run our program, there are two environment variables that we can set to control the behavior of PCAST. The first is `PGI_COMPARE`. This environment variable contains a comma-separated list of options that control various parameters of the comparison. You can, for example, set relative or absolute tolerance thresholds, halt at the first difference found, and more. See the User's Guide for a full listing of the available options.

The second option is `PGI_ACC_DEBUG` which is specific to autocompare. It simply turns on more verbose debugging as to what and where the host is comparing.

```
$ PGI_COMPARE=summary,rel=1 PGI_ACC_DEBUG=0x10000 ./a.out
comparing a1 in example.c, function main line 26
comparing a2 in example.c, function main line 26
compared 2 blocks, 2000 elements, 8000 bytes
no errors found
relative tolerance = 0.100000, rel=1
```

Autocompare will automatically detect all data differences at execution of the appropriate data directives.

You can explicitly compare data with the `acc_compare` function. When called, it will compare data in GPU memory with the corresponding data in CPU memory. It must be called from CPU code, not an OpenACC compute region. Therefore, you must compile with `-ta=tesla:redundant`.

For reference, `acc_compare`'s signature is the following:

```
acc_compare(x, n)
```

Where `x` is the data to compare and `n` is the number of elements to compare. Note here that the number of elements to compare is not sized in bytes. In our example we want to compare `size` elements even though `size` is an integer. The call would remain the same even if we changed the type from `int` to, say, `double`.

```
#pragma acc data copy(a1[0:size], a2[0:size])
{
    for (t = 0; t < 5; t++) {
        #pragma acc parallel
        for(i = 0; i < size; i++) {
            a2[i] += a1[i];
        }
        acc_compare(a2, size);
    }
}
```

```
    }
}
```

Compile with the following command, noting the redundant flag:

```
$ pgcc -Minfo=accel -ta=tesla:redundant -o a.out example.c
```

```
$ PGI_COMPARE=summary,rel=1 PGI_ACC_DEBUG=0x10000 ./a.out
compared 5 blocks, 5000 elements, 20000 bytes
no errors found
relative tolerance = 0.100000, rel=1
```

Note that we're calling `acc_compare` five times on an array of size 1000, with each element of size four bytes, totalling 20,000 bytes. With `autocompare` the data was compared at the end of the data directive instead of the end of the outer loop.

While `acc_compare` will compare the contents of the data in memory, `pgi_compare` writes the data to be compared to a file. Subsequent calls to `pgi_compare` will compare data between the file and data in the host memory. One advantage to this approach is that successive comparisons can be done in a quicker fashion since a "golden" copy is already in the file. The downside to this approach, however, is that the data file can grow very large depending on the amount of data the program is utilizing. In general, it is a good idea to use `pgi_compare` on programs where the data size is relatively small.

Its signature is as follows, where `a` is the variable to be compared, `"type"` is a string of the variable, `n` is the number of elements to be compared and the last two arguments specify function name and line number respectively:

```
pgi_compare(a, "type", n, "str", int)
```

```
#pragma acc data copy(a1[0:size], a2[0:size])
{
    for (t = 0; t < 5; t++) {
        #pragma acc parallel
        #pragma acc update host(a2[0:size])
        for(i = 0; i < size; i++) {
            a2[i] += a1[i];
        }

        pgi_compare(a2, "float", size, "main", 23);
    }
}
```

Compiling with `redundant` or `autocompare` options are not required to use `pgi_compare`. Running the code yields the following:

```
$ PGI_COMPARE=summary,rel=1 ./a.out
datafile pgi_compare.dat created with 5 blocks, 5000 elements, 20000 bytes
$ PGI_COMPARE=summary,rel=1 ./a.out
datafile pgi_compare.dat compared with 5 blocks, 5000 elements, 20000 bytes
no errors found
relative tolerance = 0.100000, rel=1
```

The first time we run the program, the data file "pgi_compare.dat" is created. Subsequent runs compare calculated data against the file. You can use the `PGI_COMPARE` environment variable to set the name of the file or force the program to create a new file with `PGI_COMPARE=create`.

For additional information, see the [PCAST page](http://pgicompilers.com/pgcast) located at pgicompilers.com/pgcast

1.9. Deep Copy Overview

True deep copy directives allow you to specify a subset of members to move between host and device memory within the declaration of the aggregate data structure, including named policies that allow distinct sets of members to be copied at different points in the program.

Usage

There are two directives for Deep Copy: `shape` and `policy`. The `shape` directive is used to define the size of dynamic data objects. This is especially useful for C/C++ struct, class, and union definitions where declarations like `my_type *ptr` include no information regarding the size of the data object accessible via the pointer. By contrast, Fortran mandates that dynamic members declare boundary information.

C/C++ signature:

```
#pragma acc shape[<shapename>] (shape-var-list) [clause-list]
```

Fortran signature:

```
!$acc shape<shapename>(shape-var-list) [clause-list]
```

Shape name is optional for C/C++, but there can be at most one unnamed shape directive for each aggregate type. The `shape-var-list` list specifies the bounds of the variables of the aggregate type for data movement. Each variable within the list must be a dynamic member defined within the corresponding type.

For the `shape` directive, there are two clauses: `init_needed(var-list)` and `type(aggregate-type-name)`. Variables that must be copied to the device via a `copy` or `copyin` clause should be specified within `init_needed(var-list)`. Such variables will be copied to the device after their memory is allocated.

The `type(aggregate-type-name)` is used to specify to which aggregate type the shape policy applies. It is only relevant if the directive appears outside of the aggregate type's definition.

The `policy` directive defines data motion between device and host. All policy directives must be named. Each directive can also specify its own shape information with the shape name. If no shape name is given, then the default shape information is used.

C/C++ signature:

```
#pragma acc policy<policy-name[: shape-name]>[clause-list]
```

Fortran signature:

```
!$acc policy<policy-name[: shape-name]>[clause-list]
```

Clauses that describe data motion (*create*, *copy*, *copyin*, *copyout*, *update*) take as arguments a list of variables defined in the aggregate type. Typically, *create* and *copyin* clauses should be used for enter data directives and *copyout* for exit data directives. For *update* directives, only the *update* clause is allowed.

To enable use of deep copy, add `-ta=tesla:deepcopy` to the compiler options used to compile your program.

Here's an example to illustrate some of these concepts. Consider the following class definition:

```
class aggr {
private:
    float* a;
    float* b;
    float* c;
    size_t len;
#pragma acc shape(a[0:len], b[0:len], c[0:len]) init_needed(len)
#pragma acc policy<create_all> create(a, b, c)
#pragma acc policy<out_all> copyout(a, b, c)

public:
    ...
}
```

Note the *shape* and *policy* directives. The code initializes pointers *a*, *b*, *c* to point to heap-allocated arrays of length *len*. We specify the shape of our aggregate data type to consist of the three pointers. The *len* member in the *init_needed* clause specifies that *len* must be initialized from the host, even if the struct variable occurs in a *create* clause.

We also create two named policies, *create_all* and *out_all* to specify how we want data to be moved between device and host. No shape information is declared in the policy since it will use the existing shape directive inside the class.

The constructor for our class is as follows:

```
aggr(size_t n) {
    len = n;
    a = new float[len];
    b = new float[len];
    c = new float[len];
}
```

We create an instance of our class and call some of its member functions:

```
aggr ag(1000);
```

```
ag.init();
ag.add();
ag.finish();
```

```
void init() {
#pragma acc enter data create(this<create_all>)

#pragma acc parallel loop present(a, b)
  for(size_t i = 0; i < len; ++i) {
    a[i] = sin(i) * sin(i);
    b[i] = cos(i) * cos(i);
  }
}
```

The `init` function uses the `create_all` policy to create the data on the device. The subsequent parallel loops initialize `a` and `b`. The important thing to note here is that we're using the policy to specify that data should be created on the device. If, for example, we omit `c` from the policy and change it to `copy` instead of `create`, then `a` and `b` would be copied to the device from the host, and `c` created on the device:

```
#pragma acc policy<create_all> copy(a, b)
...
#pragma acc enter data create(this<create_all>)
```

The remaining code assigns the results of $\sin^2 + \cos^2$ to `c` and copies the data back to the host as per the `out_all` policy.

```
void add() {
#pragma acc parallel loop present(a, b, c)
  for(size_t i = 0; i < len; ++i) {
    c[i] = a[i] + b[i];
  }
}

void finish() {
#pragma acc exit data copyout(this<out_all>)
}

~aggr() {
  delete a;
  delete b;
  delete c;
}
```

Limitations

There are a few limitations with deep copy that are worth discussing. Pointers and array boundary definitions in the directives must be 32 or 64 bit integer values.

Any expression that evaluates to a boundary or length must be one of the following forms:

- ▶ `A + B`
- ▶ `A - B`

- ▶ $A * B$
- ▶ A / B
- ▶ $A*B + C$
- ▶ $A*B - C$
- ▶ $A/B + C$
- ▶ $A/B - C$

Where A , B and C must be either constant values or integer members. No parentheses are allowed in these expressions.

Run-time issues can arise if two or more pointers from the same aggregate type point to memory locations that overlap, but only if the pointers differ in size. For example, if pointer A of some size S_1 pointed to memory that overlapped with pointer B of size S_2 where $S_1 < S_2$ and A is defined earlier than B , then there is a risk of run-time errors. The simple solution is to swap the order of declaration so B is defined before A .

Data structures that have some kind of circular structure (i.e. a descendant pointer pointing to a parent or ancestor structure) is currently not supported in this implementation of deep copy. In these cases it is necessary to manually manage the movement of data per application-specific needs.

`-Mipa`, `-Minline`, `-ta=tesla:managed` may conflict with `-ta=tesla:deepcopy` in this implementation. This will be resolved in future releases.

Additional Information

For an in-depth discussion of PGI's deep copy implementation, see Michael Wolfe's [True Deep Copy Beta Feature in PGI 18.7 Compilers](https://www.pgroup.com/blogs/posts/deep-copy-beta.htm) blog post located at <https://www.pgroup.com/blogs/posts/deep-copy-beta.htm>. Registration is required.

1.10. C++ Compiler

1.10.1. C++17

The PGI 18.1 C++ compiler introduces partial support for the C++17 language standard; access this support by compiling with `--c++17` or `-std=c++17`.

Supported C++17 core language features are available on Linux (requires GCC 7 or later) and OS X.

This PGI compiler release supports the following C++17 language features:

- ▶ Structured bindings
- ▶ Selection statements with initializers
- ▶ Compile-time conditional statements, a.k.a. `constexpr if`
- ▶ Fold expressions
- ▶ Inline variables

- ▶ Constexpr lambdas
- ▶ Lambda capture of `*this` by value

The following C++17 language features are not supported in this release:

- ▶ Class template deduction
- ▶ Auto non-type template parameters
- ▶ Guaranteed copy elision

The PGI products do not include a C++ standard library, so support for C++17 additions to the standard library depends on the C++ library provided on your system. On Linux, GCC 7 is the first GCC release with significant C++17 support. On OS X, there is no support for any of the C++17 library changes with one exception: `std::string_view` is available on OS X High Sierra.

The following C++ library changes are supported when building against GCC 7:

- ▶ `std::string_view`
- ▶ `std::optional`
- ▶ `std::variant`
- ▶ `std::any`
- ▶ Variable templates for metafunctions

The following C++ library changes are not available on any system that this PGI release supports:

- ▶ Parallel algorithms
- ▶ Filesystem support
- ▶ Polymorphic allocators and memory resources

1.10.2. C++ and OpenACC

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

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

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

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

CUDA Toolkit Version	Maximum GNU Version Supported
CUDA 7.5	GCC 4.9
CUDA 8.0	GCC 5.x
CUDA 9.0	GCC 6.x
CUDA 9.1	GCC 6.x
CUDA 9.2	GCC 7.x

1.10.3. C++ Compatibility

Optional packages that provide a C++ interface, such as the MPI package included with all PGI products, require the use of the `pgc++` compiler driver for compilation and linking.

These optional packages include:

- ▶ ESMF
- ▶ MPICH
- ▶ MVAPICH
- ▶ NetCDF
- ▶ Open MPI
- ▶ Parallel NetCDF
- ▶ ScaLAPACK

1.11. Runtime Library Routines

PGI 2018 supports runtime library routines associated with the PGI Accelerator compilers. For more information, refer to *Using an Accelerator* in the PGI Compiler User's Guide.

1.12. Library Interfaces

The PGI products contain a number of libraries that export C interfaces by using Fortran modules. These libraries and functions are described in the PGI Compiler User's Guide.

1.13. Environment Modules

On Linux, if you use the Environment Modules package (e.g., the `module load` command), then PGI 2018 includes a script to set up the appropriate module files.

1.14. LLVM Code Generator

PGI 2018 includes a production version of the PGI Linux/x86-64 compilers with an LLVM code generator and OpenMP runtime. It includes most features, languages, and programming model support found in the default PGI compilers, as well as the PGI debugger and profiler.

You have a few options for using the PGI compilers with the LLVM code generator including the following:

Compiler Option: `-Mllvm`

If you set your path to include the location of the default compilers, `linux86-64/18.7/bin`, then the compilation option `-Mllvm` will invoke the LLVM back end compilers in `linux86-64-llvm/18.7/bin`.

Environment Modules: `pgi-llvm`

If your environment is set up to use PGI's environment module files, the following commands load the PGI compilers with the default code generator:

```
module load pgi/18.7
```

By contrast, to load the PGI compilers with the LLVM code generator, use this set of commands:

```
module load pgi-llvm
module load pgi/18.7
```

To clear your environment, you can use:

```
module purge
```

Environment variable: `PATH`

Set your path to include `linux86-64-llvm/18.7/bin`.

Take care not to mix object files compiled with the default PGI compilers and the PGI compilers with the LLVM backend. While the generated code is compatible, the OpenMP runtime libraries are not.

Features available in the PGI Linux/x86-64 compilers using the LLVM code generator:

- ▶ Optimized OpenMP atomics.
- ▶ OpenMP 4.5 features, not including GPU offload.
- ▶ Improved performance for some applications compared to the default PGI x86-64 code generator.

Limitations of the PGI Linux/x86-64 compilers with the LLVM code generator:

- ▶ Only available for Linux; not available for macOS or Windows.
- ▶ Fortran debugging is available only for breakpoints, call stacks, and basic types.
- ▶ PGI Unified Binary is not available.
- ▶ Interprocedural optimization using the `-Mipa` option is not available.
- ▶ Some source-code directives, e.g. `DEC$`, may not have an effect.

Chapter 2.

RELEASE OVERVIEW

This chapter provides an overview of Release 2018 of the PGI Accelerator™ C11, C++14 and Fortran 2003 compilers and development tools for 64-bit x86-compatible processor-based workstations, servers, and clusters running versions of the Linux, Apple macOS and Microsoft Windows operating systems.

2.1. Licensing

All PGI products include exactly the same PGI compilers and tools software. The difference is in which features are enabled by the license keys.

PGI release 2018 version 18.1 and newer contain updated (v11.14.1.3) [FlexNet Publisher](#) license management software.

This FlexNet update addresses a [security vulnerability](#) on Windows, [license borrowing](#) ('[Imborrow](#)') issues and other improvements.



Important Users with PGI 2017 (17.x) or older need to update their license daemons to support 18.1 or newer. The new license daemons are backward-compatible with older PGI releases. For more information, see the [FlexNet Update FAQ](#).

2.1.1. Licensing Terminology

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

- ▶ **License** – the right to use PGI compilers and tools as defined by the End-user License Agreement (EULA), this is a legal agreement between NVIDIA and PGI end-users. PGI Professional (for-fee, perpetual) licenses are identified by a Product Identification Number (PIN - see below). You can find a copy of the EULA on the [PGI website](#), [pgicompilers.com/LICENSE](#), and in the `$PGI/<platform>/<rel_number>/doc` 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. For PGI Professional, License keys are generated by each PGI end-user on the PGI website 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** – Product Identification Number, a unique 6-digit number associated with a PGI Professional license. This PIN is included in your order confirmation. The PIN can also be found in your license key file after **VENDOR_STRING=**.
- ▶ **PIN tie code** – A unique 16-digit number associated with each license (PIN) that allows others to "tie" that license to their [PGI user account](https://pgicompilers.com/), pgicompilers.com/ account for administrative purposes. PGI Professional licensees can use their PIN tie code to share license administration capabilities with others in their organization.

2.1.2. Bundled License Key

Installation may place a temporary license key file named `license.dat` in the PGI installation directory if no such file already exists.

If you use a separate license server, for example

`LM_LICENSE_FILE=port@server.domain.com`, that supports this version, it is recommended that you remove or rename the license key file in the installation directory.

2.1.3. Node-locked and Network Floating Licenses

- ▶ Node-locked single-user licenses allow one user at a time to compile solely on the system on which both the PGI compilers and tools, and PGI license server are installed.
- ▶ Network floating licenses allow one or more users to use the PGI compilers and tools concurrently on any compatible client systems networked to a license server, that is, the system on which the PGI network floating license key(s) are installed. There can be multiple installations of the PGI compilers and tools on client systems connected to the license server; and client systems can use the license concurrently up to the maximum number of seats licensed for the license server.

2.2. Release Components

Release 2018 includes the following components:

- ▶ PGFORTRAN™ native OpenMP and OpenACC Fortran 2003 compiler.
- ▶ PGCC® native OpenMP and OpenACC ANSI C11 and K&R C compiler.
- ▶ PGC++® native OpenMP and OpenACC ANSI C++14 compiler.
- ▶ PGI Profiler® OpenACC, CUDA, OpenMP, and multi-thread graphical profiler.
- ▶ PGI Debugger® MPI, OpenMP, and multi-thread graphical debugger.
- ▶ Open MPI version 2.1.2 for 64-bit Linux including support for NVIDIA GPUDirect. Note that 64-bit linuxx86-64 MPI messages are limited to < 2 GB size each. As NVIDIA GPUDirect depends on InfiniBand support, Open MPI is also configured to use InfiniBand hardware if it is available on the system. InfiniBand support requires OFED 3.18 or later.
- ▶ MPICH libraries, version 3.2, for 64-bit macOS development environments.

- ▶ ScaLAPACK 2.0.2 linear algebra math library for distributed-memory systems for use with Open MPI, MPICH or MVAPICH, and the PGI compilers on 64-bit Linux and macOS for Intel 64 or AMD64 CPU-based installations.
- ▶ Microsoft HPC Pack 2012 MS-MPI Redistributable Pack (version 4.1) for 64-bit development environments (Windows only).
- ▶ BLAS and LAPACK library based on the customized OpenBLAS project source.
- ▶ A UNIX-like shell environment for 64-bit Windows platforms.
- ▶ FlexNet license utilities.
- ▶ Documentation in man page format and [online](https://www.pgicompilers.com/docs), [pgicompilers.com/docs](https://www.pgicompilers.com/docs), in both HTML and PDF formats.

2.2.1. Additional Components

PGI floating license holders may download additional components for Linux from the PGI website including:

- ▶ MPICH MPI libraries
- ▶ MVAPICH2 MPI libraries

2.2.2. MPI Support

You can use PGI products to develop and debug MPI applications. PGI node-locked licenses support debugging of up to 16 local MPI processes. PGI network floating licenses provide the ability to debug up to 256 local or remote MPI processes.

2.3. 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 [PGI online glossary](https://www.pgicompilers.com/definitions) located at [pgicompilers.com/definitions](https://www.pgicompilers.com/definitions).

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

Intel 64

64-bit Intel x86-64 CPUs including Intel Core processors, Intel Xeon Nehalem, Sandy Bridge, Ivy Bridge, Haswell, Broadwell and Skylake processors, and Intel Xeon Phi Knights Landing.

AMD64

64-bit AMD™ x86-64 CPUs including Bulldozer, Piledriver and EPYC processors.

2.4. Supported Platforms

There are three platforms supported by the PGI compilers and tools for x86-64 processor-based systems.

- ▶ **64-bit Linux** – supported on 64-bit Linux operating systems running on a 64-bit x86 compatible processor.

- ▶ **64-bit macOS** – supported on 64-bit Apple macOS operating systems running on a 64-bit Intel processor-based Macintosh computer.
- ▶ **64-bit Windows** – supported on 64-bit Microsoft Windows operating systems running on a 64-bit x86-compatible processor.

2.5. Supported Operating System Updates

This section describes updates and changes to PGI 2018 that are specific to Linux, macOS, and Windows.

2.5.1. Linux

- ▶ CentOS 6.4 through 7.4
- ▶ Fedora 14 through 27
- ▶ openSUSE 13.2 through openSUSE Leap 42.3
- ▶ RHEL 6.4 through 7.4
- ▶ SLES 12 through SLES 12 SP 3
- ▶ Ubuntu 14.04, 16.04, 17.04, 17.10

2.5.2. Apple macOS

PGI 2018 for macOS supports most of the features of the version for Linux environments. Except where noted in these release notes or the user manuals, the PGI compilers and tools on macOS function identically to their Linux counterparts.

- ▶ The compilers, debugger, and profiler are supported on macOS versions 10.10.5 (Yosemite) through 10.13 (High Sierra).

2.5.3. Microsoft Windows

PGI products for Windows support most of the features of the PGI products for Linux environments. PGI products on all Windows systems include the Microsoft Open Tools but also require that a Microsoft Windows Software Development Kit (SDK) be installed prior to installing the compilers.



PGI 18.7 is the last release for Windows that includes bundled Microsoft toolchain components. Future releases will require users to have the Microsoft toolchain components pre-installed on their systems.



PGI 2018 requires the Windows 10 SDK, even on Windows 7 and 8.1.

These Windows operating systems are supported in PGI 2018:

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

- ▶ Windows 10
- ▶ Windows Server 2012
- ▶ Windows Server 2016

2.6. CUDA Toolkit Versions

The PGI compilers use NVIDIA's CUDA Toolkit when building programs for execution on an NVIDIA GPU. Every PGI installation package puts the required CUDA Toolkit components into a PGI installation directory called `2018/cuda`.

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

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

PGI 18.7 includes the following versions of the CUDA Toolkit:

- ▶ CUDA 9.1
- ▶ CUDA 9.2

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

If you do not specify a version of the CUDA Toolkit, the compiler uses the version of the CUDA Driver installed on the system on which you are compiling to determine which CUDA Toolkit to use. This auto-detect feature is new in the PGI 18.7 release; auto-detect is especially convenient when you are compiling and running your application on the same system. Here is how it works. In the absence of any other information, the compiler will look for a CUDA Toolkit version in the PGI `2018/cuda` directory that matches the version of the CUDA Driver installed on the system. If a match is not found, the compiler searches for the newest CUDA Toolkit version that is not newer than the CUDA Driver version. If there is no CUDA Driver installed, the PGI 18.7 compilers fall back to the default of CUDA 9.1. Let's look at some examples.

If the only PGI compiler you have installed is PGI 18.7, then

- ▶ If your CUDA Driver is 9.2, the compilers use CUDA Toolkit 9.2.
- ▶ If your CUDA Driver is 9.1, the compilers use CUDA Toolkit 9.1.
- ▶ If your CUDA Driver is 9.0, the compilers will issue an error that CUDA Toolkit 9.0 was not found; CUDA Toolkit 9.0 is not bundled with PGI 18.7.
- ▶ If you do not have a CUDA driver installed on the compilation system, the compilers use CUDA Toolkit version 9.1.
- ▶ If your CUDA Driver is newer than CUDA 9.2, the compilers will still use the CUDA Toolkit 9.2. The compiler selects the newest CUDA Toolkit it finds that is not newer than the CUDA Driver.

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

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

```
pgcc -ta=tesla:cuda9.2
```

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

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

```
set DEFCUDAVERSION=9.2;
```

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

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

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

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

To use a CUDA toolkit that is not installed with a PGI release, such as CUDA 8.0 with PGI 18.7, there are three options:

- ▶ Use the rcfile variable `DEFAULT_CUDA_HOME` to override the base default

```
set DEFAULT_CUDA_HOME = /opt/cuda-8.0;
```

- ▶ Set the environment variable `CUDA_HOME`

```
export CUDA_HOME=/opt/cuda-8.0
```

- ▶ Use the compiler compilation line assignment `CUDA_HOME=`

```
pgfortran CUDA_HOME=/opt/cuda-8.0
```

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

1. If you do not tell the compiler which CUDA Toolkit version to use, the compiler picks the CUDA Toolkit from the PGI installation directory `2018/cuda` that matches the version of the CUDA Driver installed on your system. If the PGI

installation directory does not contain a direct match, the newest version in that directory which is not newer than the CUDA driver version is used. If there is no CUDA driver installed on your system, the compiler falls back on an internal default; in PGI 18.7, this default is CUDA 9.1.

2. The `rcfile` variable `DEFAULT_CUDA_HOME` will override the base default.
3. The environment variable `CUDA_HOME` will override all of the above defaults.
4. The environment variable `PGI_CUDA_HOME` overrides all of the above; it is available for advanced users in case they need to override an already-defined `CUDA_HOME`.
5. A user-specified `cudaX.Y` sub-option to `-Mcuda` and `-ta=tesla` will override all of the above defaults and the CUDA Toolkit located in the PGI installation directory `2018/cuda` will be used.
6. The compiler compilation line assignment `CUDA_HOME=` will override all of the above defaults (including the `cudaX.Y` sub-option).

2.7. Compute Capability

The compilers can generate code for NVIDIA GPU compute capabilities 3.0 through 7.0. The compilers construct a default list of compute capabilities that matches the compute capabilities supported by the GPUs found on the system used in compilation. If there are no GPUs detected, the compilers select `cc35`, `cc50`, `cc60`, and `cc70`.

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

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

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

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

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

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

2.8. Precompiled Open-Source Packages

Many open-source software packages have been ported for use with PGI compilers on Linux x86-64.

The following PGI-compiled open-source software packages are included in the PGI Linux x86-64 download package:

- OpenBLAS 0.2.19 – customized BLAS and LAPACK libraries based on the OpenBLAS project source.

- ▶ Open MPI 2.1.2 – open-source MPI implementation.
- ▶ ScaLAPACK 2.0.2 – a library of high-performance linear algebra routines for parallel distributed memory machines. ScaLAPACK uses Open MPI 2.1.2.

The following list of open-source software packages have been precompiled for execution on Linux x86-64 targets using the PGI compilers and are available to download from the [PGI website](https://www.pgroup.com/pgi-website) at [pgicompilers.com/downloads](https://www.pgroup.com/pgi-website/pgicompilers.com/downloads).

- ▶ MPICH 3.2 – open-source MPI implementation.
- ▶ MVAPICH2 2.2 – open-source MPI implementation.
- ▶ ESMF 7.0.2 for Open MPI 2.1.2 – The Earth System Modeling Framework for building climate, numerical weather prediction, data assimilation, and other Earth science software applications.
- ▶ ESMF 7.0.2 for MPICH 3.2.
- ▶ ESMF 7.0.2 for MVAPICH2 2.2.
- ▶ NetCDF 4.5.0 for C++11 – A set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data, written in C. Included in this package are the following components:
 - ▶ NetCDF-C++ 4.3.0 – C++ interfaces to NetCDF libraries.
 - ▶ NetCDF-Fortran 4.4.4 – Fortran interfaces to NetCDF libraries.
 - ▶ HDF5 1.10.1 – data model, library, and file format for storing and managing data.
 - ▶ CURL 7.46.0 – tool and a library (usable from many languages) for client-side URL transfers.
 - ▶ SZIP 2.1.1 – extended-Rice lossless compression algorithm.
 - ▶ ZLIB 1.2.11 – file compression library.
- ▶ NetCDF 4.4.1.1 for C++98 – includes all the components listed in NetCDF for C++11 above.
- ▶ Parallel NetCDF 1.9.0 for MPICH 3.2
- ▶ Parallel NetCDF 1.9.0 for MVAPICH2 2.2
- ▶ Parallel NetCDF 1.9.0 for Open MPI 2.1.2

In addition, these software packages have also been ported to PGI on Linux x86-64 but due to licensing restrictions, they are not available in binary format directly from PGI. You can find instructions for building them in the [Porting & Tuning Guides](#) section of the PGI website at pgicompilers.com/tips.

- ▶ FFTW 2.1.5 – version 2 of the Fast Fourier Transform library, includes MPI bindings built with Open MPI 2.1.2.
- ▶ FFTW 3.3.7 – version 3 of the Fast Fourier Transform library, includes MPI bindings built with Open MPI 2.1.2.

For additional information about building these and other packages, please see the [Porting & Tuning Guides](https://pgi.com/resources/porting-tuning-guides) section of the PGI website at pgi.com/resources/porting-tuning-guides.

2.9. Getting Started

By default, the PGI 2018 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 the aggregate option `-fast`.

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. They enable vectorization with SSE instructions, cache alignment, and flushz.



The content of the `-fast` option is host-dependent.

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

Table 3 Typical `-fast` Options

Use this option...	To do this...
<code>-O2</code>	Specifies a code optimization level of 2.
<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` also typically includes the options shown in the following table:

Table 4 Additional `-fast` Options

Use this option...	To do this...
<code>-Mvect=simd</code>	Generates packed SSE and AVX instructions.
<code>-Mcache_align</code>	Aligns long objects on cache-line boundaries.
<code>-Mflushz</code>	Sets flush-to-zero mode.



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

In addition to `-fast`, 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, -Mpf, -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 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 Linux and Windows. On Windows, PGI also supplies Microsoft redistributable files.

3.1.1. PGI Redistributables

The PGI 2018 Release includes these directories:

```
$PGI/linux86-64/18.7/REDIST
$PGI/win64/18.7/REDIST
```

These directories contain all of the PGI Linux runtime library shared object files or Windows dynamically linked libraries that can be re-distributed by PGI 2018 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 18.7 `doc` 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 2018 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, please see the [PGI frequently asked questions \(FAQ\)](#) webpage.

4.1. Platform-specific Issues

4.1.1. 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.

4.1.2. Apple macOS

The following are known issues on Apple macOS:

- ▶ The PGI 2018 compilers do not support static linking of binaries. For compatibility with future Apple updates, the compilers only support dynamic linking of binaries.

4.1.3. Microsoft Windows

The following are known issues on Windows:

- ▶ 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.

- ▶ 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 work around this problem, set **SHELL** to refer to a shell in the Cygwin `bin` directory, e.g., `/bin/bash`.

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

4.2. Issues Related to Debugging

The following are known issues in the PGI debugger:

- ▶ Debugging of PGI Unified Binaries, that is, programs built with more than one `-tp` option, is not fully supported. The names of some subprograms are modified in compilation and the debugger does not translate these names back to the names used in the application source code.
- ▶ When debugging on the Windows platform, the Windows operating system times out **stepi/nexti** operations when single stepping over blocked system calls.

4.3. Profiler-related Issues

Some specific issues related to the PGI Profiler:

- ▶ The Profiler relies on being able to directly call `'dlsym'`. If this system call is intercepted by the program being profiled or by some other library the profiler may hang at startup. We have encountered this specific problem with some implementations of MPI. We recommend you disable any features that may be intercepting the `'dlsym'` system call or disable CPU profiling with the `--cpu-profiling off` option.
 - ▶ To disable `'dlsym'` interception when using IBM's spectrum MPI set the environment variable: `PAMI_DISABLE_CUDA_HOOK=1`, omit the following option: `-gpu` and add the options: `-x PAMI_DISABLE_CUDA_HOOK` and `-disable_gpu_hooks`.

4.4. OpenACC Issues

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

ACC routine directive limitations

- ▶ Fortran assumed-shape arguments are not yet supported.

Clause Support Limitations

- ▶ Not all clauses are supported after the `device_type` clause.

Chapter 5.

CONTACT INFORMATION

You can contact PGI at:

9030 NE Walker Road, Suite 100
Hillsboro, OR 97006

Or electronically using any of the following means:

Fax: +1-503-682-2637

Sales: sales@pgroup.com

WWW: <https://www.pgroup.com> or pgicompilers.com

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

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

Submit support requests using the [PGI Technical Support Request](#) form, pgicompilers.com/support-request.

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

NVIDIA, the NVIDIA logo, Cluster Development Kit, PGC++, PGCC, PGDBG, PGF77, PGF90, PGF95, PGFORTRAN, PGHPF, PGI, PGI Accelerator, PGI CDK, PGI Server, PGI Unified Binary, PGI Visual Fortran, PGI Workstation, PGPROF, PGROUP, PVF, and The Portland Group are trademarks and/or 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

© 2013-2018 NVIDIA Corporation. All rights reserved.