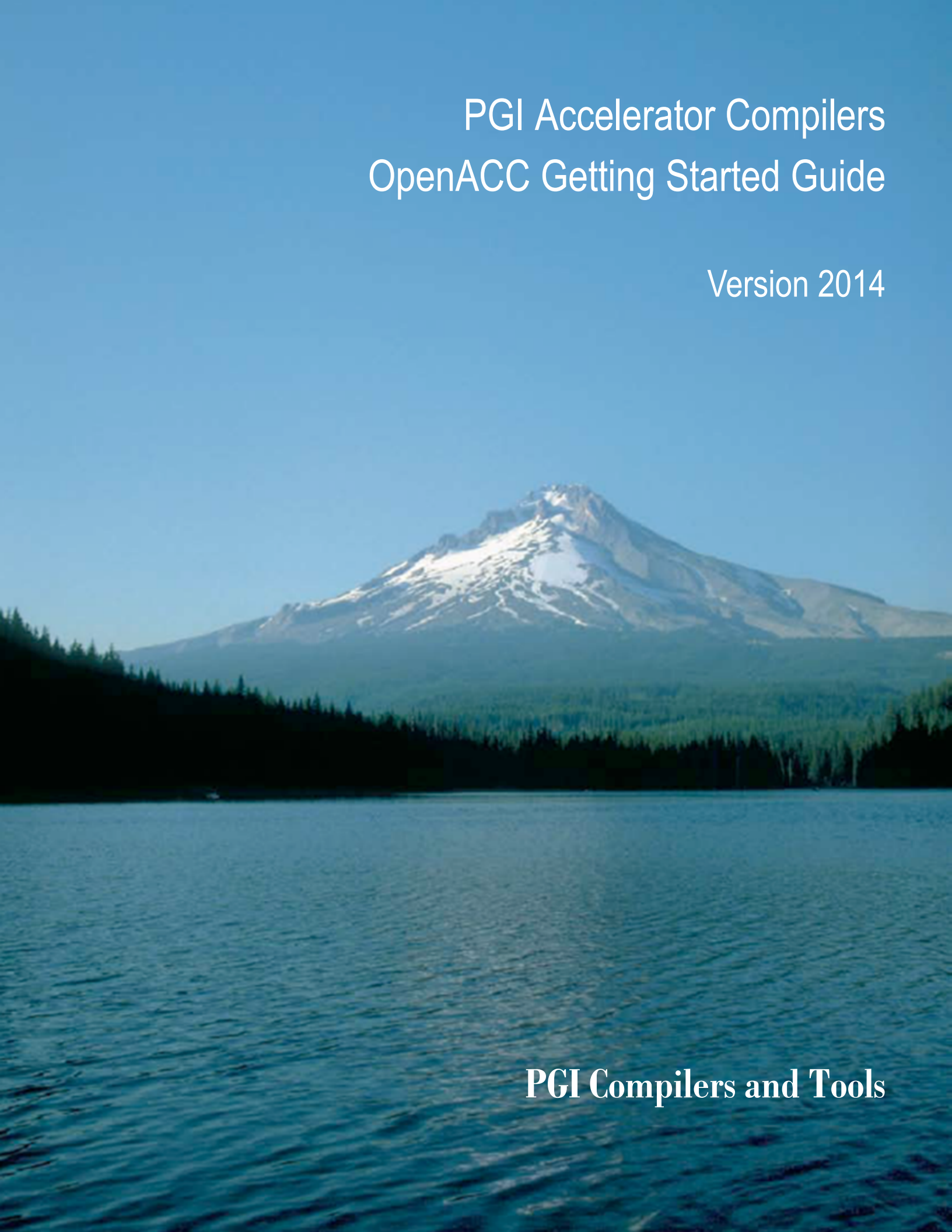
PGI Accelerator Compilers
OpenACC Getting Started Guide

Version 2014



PGI Compilers and Tools

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1. OVERVIEW

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

This Getting Started guide helps you prepare your system for using the PGI OpenACC implementation, and provides examples of how to write, build and run programs using the OpenACC directives. More information about the PGI OpenACC implementation is available at http://www.pgroup.com/openacc.

1.1. Terms and Definitions

Throughout this document certain terms have very specific meaning:

- OpenACC is a parallel programming standard describing a set of compiler directives which can be applied to standard C, C++, and Fortran to specify regions of code for offloading from a host CPU to an attached accelerator.
- A directive is, in C, a #pragma, or, in Fortran, a specially formatted comment statement that is interpreted by a compiler to augment information about or specify the behavior of the program. This document uses the term directives for either Fortran directive comments or C/C++ pragmas. Features specific to "Fortran directives" and "C pragmas" are called out as such.
- PGCC, PGC++, and PGFORTRAN are the names of the PGI compiler products.
- pgcc and pgfortran are the names of the PGI compiler drivers. pgfortran may also be spelled pgf90 and pgf95. The PGI C++ compilers are named pgcpp and pgc++. pgcpp is the driver on Windows and OS X, and uses legacy name mangling on Linux. pgc++ is the driver on Linux which uses GNU-compatible naming conventions.
- CUDA stands for Compute Unified Device Architecture; the CUDA environment from NVIDIA is a C–like programming environment used to explicitly control and program an NVIDIA GPU. OpenCL is the Open Compute Language, a standard C-like programming environment similar to CUDA that enables portable low-level general-purpose programming on GPUs and other accelerators. This programming language and model is supported by AMD for their GPUs.

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

1.2. System Prerequisites

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

- A 32-bit or 64-bit Intel or AMD x86 system running Linux, Microsoft Windows, or Apple OS X. Information about the PGI-supported releases is available at http://www.pgroup.com/support/install.htm.
- For targeting GPUs:
  - **NVIDIA:** A CUDA-enabled NVIDIA GPU and an installed driver. For NVIDIA CUDA, the driver should be version 5.5 or later. (http://www.nvidia.com/cuda).
  - **AMD:** An OpenCL-enabled AMD GPU, and the AMD OpenCL drivers, version 13.30 or later(http://www.amd.com/drivers)

1.3. Prepare Your System

To enable OpenACC, follow these steps:

1. Download the latest 14.7 packages from the Download page on the PGI website at http://www.pgroup.com/support/downloads.php .
2. Install the downloaded package.
3. Put the installed bin directory on your path.
4. Run `pgaccelinfo` to see that your GPU and drivers are properly installed and available. For NVIDIA, you should see output that looks something like the following:

```
CUDA Driver Version:  6000
NVRM version: NVIDIA UNIX x86_64 Kernel Module 331.49 Wed Feb 12
20:42:50 PST 2014
CUDA Device Number:                    0
Device Name:     Tesla K20c
Device Revision Number:   3.5
Global Memory Size:    5032706048
Number of Multiprocessors:   13
Number of SP Cores:    2496
Number of DP Cores:   832
Concurrent Copy and Execution:   Yes
Total Constant Memory:    65536
Total Shared Memory per Block:   49152
Registers per Block:    65536
Warp Size:     32
Maximum Threads per Block:   1024
Maximum Block Dimensions:   1024, 1024, 64
Maximum Grid Dimensions:   2147483647 x 65535 x 65535
Maximum Memory Pitch:    2147483647B
Texture Alignment:    512B
Clock Rate:     705 MHz
Execution Timeout:    No
Integrated Device:    No
Can Map Host Memory:    Yes
Compute Mode:    default
Concurrent Kernels:    Yes
ECC Enabled:     Yes
Memory Clock Rate:    2600 MHz
Memory Bus Width:    320 bits
L2 Cache Size:     1310720 bytes
Max Threads Per SMP:   2048
Async Engines:     2
Unified Addressing:    Yes
Initialization time:    1487991 microseconds
Current free memory:    4952023040
Upload time (4MB):    942 microseconds ( 708 ms pinned)
Download time:     1060 microseconds ( 673 ms pinned)
Upload bandwidth:    4452 MB/sec (5924 MB/sec pinned)
Download bandwidth:    3956 MB/sec (6232 MB/sec pinned)
PGI Compiler Option:   -ta=tesla:cc35
```

5. For AMD, you should see output that looks something like the following:

```
OpenCL Platform:                AMD Accelerated Parallel Processing
OpenCL Vendor:                  Advanced Micro Devices, Inc.

Device Number:                  0
Device Name:                    Tahiti
Available:                      Yes
Compiler Available:             Yes
Board Name:                     ATI FirePro V (FireGL V) Graphics Adapter
Device Version:                 OpenCL 1.2 AMD-APP (1359.4)
Global Memory Size:             3079667712
Maximum Object Size:            1073741824
Global Cache Size:              16384
Free Memory:                    3007650000
Max Clock (MHz):                950
Compute Units:                  28
SIMD Units:                     4
SIMD Width:                     16
GPU Cores:                      1792
Wavefront Width:                64
Constant Memory Size:           65536
Local Memory Size:              32768
Workgroup Size:                 256
Address Bits:                   32
ECC Support:                    No
PGI Compiler Option:            -ta=radeon:tahiti
```

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

1.4. Supporting Documentation and Examples

You may want to consult the latest OpenACC 2.0 specification, included with this release, for additional information. It is also available at the OpenACC website. Simple examples appear in Using OpenACC with the PGI Compilers.

Source code is included with this release as well in `/opt/pgi/[os][-64]/2014/examples/openacc/`

Chapter 2.
USING OPENACC WITH THE PGI COMPILERS

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

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

This release includes partial support for the OpenACC 2.0 specification. Refer to Implemented Features for details about which features are supported in this release, and what features are coming in updates over the next few months.

2.1. C Examples

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

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

int main( int argc, char* argv[] ){
    int n; /* vector length */
    float * a; /* input vector 1 */
    float * b; /* input vector 2 */
    float * r; /* output vector */
    float * e; /* expected output values */
    int i, errs;
    if( argc > 1 ) n = atoi( argv[1] );
    else n = 100000; /* default vector length */
    if( n <= 0 ) n = 100000;
    a = (float*)malloc( n*sizeof(float) );
    b = (float*)malloc( n*sizeof(float) );
    r = (float*)malloc( n*sizeof(float) );
    e = (float*)malloc( n*sizeof(float) );
    for( i = 0; i < n; ++i ){
         a[i] = (float)(i+1);
         b[i] = (float)(1000*i);
    }
    /* compute on the GPU */
    vecaddgpu( r, a, b, n );
    /* compute on the host to compare */
    for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
    /* compare results */
    errs = 0;
    for( i = 0; i < n; ++i ){
     if( r[i] != e[i] ){
     ++errs;
       }
}
printf( "%d errors found\n", errs );
return errs;
}
```

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

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

```
0 errors found
```

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

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

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

```
launch CUDA kernel file=/user/guest/a1.c function=vecaddgpu
line=6 device=0 grid=782 block=128
0 errors found
```

The extra output tells you that the program launched a kernel for the loop at line 6, with a CUDA grid of size 391, and a thread block of size 256. If you set the environment variable PGI_ACC_NOTIFY to 3, the output will include information about the data transfers as well:

```
upload CUDA data file=/user/guest/a1.c function=vecaddgpu
line=5 device=0 variable=b bytes=400000
upload CUDA data file=/user/guest/a1.c function=vecaddgpu
line=5 device=0 variable=a bytes=400000
launch CUDA kernel file=/user/guest/a1.c function=vecaddgpu
line=6 device=0 grid=782 block=128
download CUDA data file=/user/guest/a1.c function=vecaddgpu
line=7 device=0 variable=r bytes=400000
0 errors found
```

If you set the environment variable PGI_ACC_TIME to 1, the runtime summarizes the time taken for data movement between the host and GPU, and computation on the GPU. On Linux, you may need to set the LD_LIBRARY_PATH environment variable to include the /opt/pgi/linux86[-64]/14.7/lib or /opt/pgi/linux86/14.7/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 this program, you might get output similar to this:

```
0 errors found

Accelerator Kernel Timing data
/user/guest/a1.c
  vecaddgpu NVIDIA devicenum=0
      time(us): 598
      5: data copyin reached 2 times
           device time(us): total=315 max=161 min=154 avg=157
      6: kernel launched 1 times
           grid: [782] block: [128]
           device time(us): total=32 max=32 min=32 avg=32
           elapsed time(us): total=41 max=41 min=41 avg=41
      7: data copyout reached 1 times
           device time(us): total=251 max=251 min=251 avg=251
```

This tells you that the program entered one accelerator region and spent a total of about 598 microseconds in that region. It copied two arrays to the device, launched one kernel and brought one array back to the host.

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

```
vecaddgpu:
       5, Generating present_or_copyout(r[0:n])
          Generating present_or_copyin(b[0:n])
          Generating present_or_copyin(a[0:n])
          Generating Tesla code
          Generating compute capability 1.0 binary
          Generating compute capability 2.0 binary
          Generating compute capability 3.0 binary
       6, Loop is parallelizable
          Accelerator kernel generated
          6, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

This tells you that the compiler generated three versions of the code, one for NVIDIA devices with compute capability 1.0 and higher (Tesla), and one for devices with compute capability 2.0 and higher (Fermi), and third for compute capability 3.0 and higher (Kepler). It also gives the *schedule* used for the loop; in this case, the schedule is `gang,vector(128)`. This means the iterations of the loop are broken into vectors of 128, and the vectors executed in parallel by SMs or compute units of the GPU.

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

```
       6, Complex loop carried dependence of '*(b)' prevents parallelization
          Complex loop carried dependence of '*(a)' prevents parallelization
          Loop carried dependence of '*(r)' prevents parallelization
          Loop carried backward dependence of '*(r)' prevents vectorization
          Accelerator scalar kernel generated
```

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

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

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

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

int main( int argc, char* argv[] ){
       int n; /* vector length */
       float * a; /* input vector 1 */
       float * b; /* input vector 2 */
       float * r; /* output vector */
       float * e; /* expected output values */
       int i, errs;

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

2.2. Fortran Examples

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

2.2.1. Vector Addition on the GPU

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

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

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

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

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

```
0 errors found
```

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

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

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

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

The extra output tells you that the program launched a kernel for the loop at line 9, with a CUDA grid of size 7813, and a thread block of size 128. If you set the environment variable `PGI_ACC_NOTIFY` to 3, the output will include information about the data transfers as well:

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

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

```
0 errors found

Accelerator Kernel Timing data
/user/guest/f1.f90
  vecaddgpu  NVIDIA  devicenum=0
        time(us): 1,971
        8: data copyin reached 2 times
             device time(us): total=1,242 max=623 min=619 avg=621
        9: kernel launched 1 times
            grid: [7813]  block: [128]
             device time(us): total=109 max=109 min=109 avg=109
            elapsed time(us): total=118 max=118 min=118 avg=118
        12: data copyout reached 1 times
             device time(us): total=620 max=620 min=620 avg=620
```

This tells you that the program entered one accelerator region and spent a total of about 2 milliseconds in that region. It copied two arrays to the device, launched one kernel and brought one array back to the host.

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

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

```
vecaddgpu:
      8, Generating present_or_copyout(r(:n))
         Generating present_or_copyin(b(:n))
         Generating present_or_copyin(a(:n))
         Generating Tesla code
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
      9, Loop is parallelizable
         Accelerator kernel generated
          9, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

This tells you that the compiler generated three versions of the code, one for NVIDIA devices with compute capability 1.0 and higher (Tesla), and one for devices with compute capability 2.0 and higher (Fermi), and one for devices with compute capability 3.0 and higher (Kepler). It also gives the schedule used for the loop; in this case, the schedule is `gang, vector(128)`. This means the iterations of the loop are broken into vectors of 128, and the vectors executed in parallel by SMPs of the GPU. This output is important because it tells you when you are going to get parallel execution or sequential execution.

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

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

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

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

2.2.2. Multi-Threaded Program Utilizing Multiple Devices

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

```
program tdot
! Compile with "pgfortran -mp -acc tman.f90 -lacml
! Compile with "pgfortran -mp -acc tman.f90 -lblas,
!   where acml is not available
! Set OMP_NUM_THREADS environment variable to run with
! up to 2 threads, currently.
!
use openacc
use omp_lib
!
integer, parameter :: N = 10000
real*8 x(N), y(N), z
integer, allocatable :: offs(:)
real*8, allocatable :: zs(:)
real*8 ddot

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

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

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

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

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

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

2.3. Troubleshooting Tips and Known Limitations

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

The Linux CUDA driver will power down an idle GPU. This means if you are using a GPU with no attached display, or an NVIDIA Tesla compute-only GPU, and there are no open CUDA contexts, the GPU will power down until it is needed. Since it may take up to a second to power the GPU back up, you may experience noticeable delays when you start your program. When you run your program with the environment variable `PGI_ACC_TIME` set to 1, this time will appear as initialization time. If you have an NVIDIA S1070 or S2050 with four GPUs, this initialization time may be up to 4 seconds. If you are running many tests, or want to isolate the actual time from the initialization time, you can run the PGI utility `pgcudainit` in the background. This utility opens a CUDA context and holds it open until you kill it or let it complete.

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

Chapter 3. IMPLEMENTED FEATURES

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

3.1. In This Release

This release includes full support for the OpenACC 1.0 specification except for the `firstprivate( )` clause for the Parallel construct. In addition, this release includes support for the following OpenACC 2.0 features:

- Procedure calls (routine directive)
- Unstructured data lifetimes
- Create and device_resident clauses for the Declare directive
- Multidimensional dynamically allocated C/C++ arrays
- Ability to call CUDA Fortran atomic functions on NVIDIA
- Complete run-time API support

3.2. Defaults

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

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

3.3. Environment Variables

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

- The names of the environment variables must be upper case.
- The values of environment variables are case insensitive and may have leading and trailing white space.
- The behavior is implementation-defined if the values of the environment variables change after the program has started, even if the program itself modifies the values.

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

Table 1 Supported Environment Variables

Use this environment variable...	To do this...
PGI_ACC_TIME	Enables a lightweight profiler to measure data movement and accelerator kernel execution time and print a summary at the end of program execution.
PGI_ACC_PROFILE	Is used by pgcollect internally to enable the lightweight PGI timers and write the information out for pgprof.
PGI_ACC_PROFLIB	Enables 3rd party tools interface using the new profiler dynamic library interface.
PGI_ACC_NOTIFY	Writes out a line for each kernel launch and/or data movement. When set to an integer value, the value, is used as a bit mask to print information about kernel launches (value 1), data transfers (value 2), region entry/exit (value 4), wait operations or synchronizations with the device (value 8), and device memory allocates and deallocates (value 16).
PGI_ACC_SYNCHRONOUS	Disables asynchronous launches and data movement.
PGI_ACC_DEVICE_NUM = = ACC_DEVICE_NUM	Sets the default device number to use. PGI_ACC_DEVICE_NUM overrides ACC_DEVICE_NUM. Controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices attached to the host.
PGI_ACC_DEVICE_TYPE = = ACC_DEVICE_TYPE = = ACC_DEVICE	Sets the default device type to use. PGI_ACC_DEVICE_TYPE overrides ACC_DEVICE_TYPE. Controls which accelerator device to use when executing accelerator regions, if the program has been compiled to use more than one different type of device. The value of this environment variable is implementation-defined, and currently may be the string NVIDIA, TESLA, RADEON, or HOST
PGI_ACC_BUFFERSIZE	For NVIDIA CUDA devices, this defines the size of the pinned buffer used to transfer data between host and device.
PGI_ACC_CUDA_GANGLIMIT	For NVIDIA CUDA devices, this defines the maximum number of gangs (CUDA thread blocks) that will be launched by a kernel.
PGI_ACC_DEV_MEMORY	For AMD GPUs, this sets the maximum buffer size ot allocate. The runtime will allocate buffers of this size, then suballocate data within these buffers.

3.4. OpenACC Fortran API Extensions

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

3.4.1. acc_malloc

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

There is one supported call format in PGI Fortran:

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

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

3.4.2. acc_free

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

There are two supported call formats in PGI Fortran:

```
subroutine acc_free ( devptr )
```

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

```
subroutine acc_free ( dev )
```

where *dev* is a CUDA Fortran device array

3.4.3. acc_map_data

The `acc_map_data` routine associates (maps) host data to device data. The first argument is a host array, contiguous host array section, or address contained in a type(c_ptr). The second argument must be a device address contained in a type(c_devptr), such as would be returned from `acc_malloc` or `acc_deviceptr`, or a CUDA Fortran device array. There are 4 supported call formats in PGI Fortran:

There are four supported call formats in PGI Fortran:

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

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

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

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

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

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

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

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

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

3.4.4. acc_unmap_data

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

There is one supported call format in PGI Fortran:

```
subroutine acc_unmap_data ( host )
```

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

3.4.5. acc_deviceptr

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

There is one supported call format in PGI Fortran:

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

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

3.4.6. acc_hostptr

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

There are two supported call formats in PGI Fortran:

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

where *dev* is a CUDA Fortran device array

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

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

3.4.7. acc_is_present

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

There are two supported call formats in PGI Fortran:

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

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

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

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

3.4.8. acc_memcpy_to_device

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

There are four supported call formats in PGI Fortran:

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

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

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

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

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

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

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

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

3.4.9. acc_memcpy_from_device

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

There are four supported call formats in PGI Fortran:

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

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

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

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

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

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

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

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

3.5. Known Limitations

This section includes the known limitations to OpenACC directives. PGI plans to support these features in a future release, though separate compilation and extern variables for Radeon will be deferred until OpenCL 2.0 is released.

3.5.1. ACC routine directive Limitations

- The `routine` directive has limited support on AMD radeon. Separate compilation is not supported on radeon, and selecting the option `-ta=radeon` disables the `rdc` suboption for `-ta=tesla`.
- Extern variables may not be used with `acc routine` procedures.
- In Fortran, only functions that return integer or real values are supported with `acc routine`.
- In C and C++, only int, float, double, or void functions are supported with `acc routine`.
- Reductions in procedures with `acc routine` are not supported.
- Fortran assumed-shape arguments are not yet supported.

3.5.2. Clause Support Limitations

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

3.5.3. Known Limitations

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

3.6. Interactions with Optimizations

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

3.6.1. Interactions with Inlining

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

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

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

3.7. In Future Releases

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

- The `deviceptr` data clause for Fortran dummy arguments.
- The `device_resident` clause on the declare directive.
- The `firstprivate()` clause on parallel regions.

Chapter 4. CONTACT INFORMATION

You can contact PGI at:

20400 NW Amberwood Drive Suite 100
Beaverton, OR 97006

Or electronically using any of the following means:

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

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

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

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

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

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

http://www.pgroup.com/support

Phone support is not currently available.

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

PGI®