PGI Accelerator™ Compilers
OpenACC Getting Started Guide

Version 12.6

The Portland Group®

OpenACC 2012 Getting Started Guide

Printed in the United States of America

First Printing: Release 2012, version 12.3, March 2012
Second Printing: Release 2012, version 12.4, April 2012
Third Printing: Release 2012, version 12.5, May 2012
Fourth Printing: Release 2012, version 12.6, July 2012

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

Contents

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 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. In particular, the whole specification is available at http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

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 PGI's OpenACC implementation is available at http://www.pgroup.com/openacc.

This release of the PGI compilers implements the OpenACC specification. In particular, where there were conflicts between the PGI Accelerator Programming directives and the OpenACC directives, this release uses the OpenACC interpretation. See Chapter 3, PGI Accelerator Model Interoperability, for examples and details.

Terminology and Definitions

Throughout this document certain terms have very specific meaning:

- OpenACC is the name of the specification, which includes compiler directives, runtime routines, and environment variables.
- PGCC 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`.
- CUDA is the parallel computing platform and programming model invented and supported by NVIDIA for GPUs.

System Prerequisites

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

- A 32-bit or 64-bit Linux, Microsoft Windows, or Apple OS/X Intel or AMD x86 system, with a PGI-supported and CUDA-supported release of the operating system. Get information about the PGI-supported Linux releases at http://www.pgroup.com/support/install.htm. Get information about CUDA-supported Linux releases at http://www.nvidia.com/cuda.
- A CUDA-enabled NVIDIA GPU.
- An installed CUDA driver, version 4.0 or later. CUDA drivers can be downloaded at http://www.nvidia.com/cuda.

Prepare Your System

To enable OpenACC, follow these steps:

1. Download the latest 12.6 Linux packages from the Download page on the PGI website at http://www.pgroup.com/.
2. Install the downloaded package.
3. Put the installed bin directory on your path.
4. Run `pgaccelinfo` to see that your NVIDIA GPU and CUDA drivers are properly installed and available. You should see output that looks something like the following:

```
CUDA Driver Version:           4010
NVRM version: NVIDIA UNIX x86_64 Kernel Module  285.05.33  Thu Jan
19 14:07:02 PST 2012

Device Number:                 0
Device Name:                   GeForce GTX 280
Device Revision Number:        1.3
Global Memory Size:            1073414144
Number of Multiprocessors:     30
Number of Cores:               240
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 16384
Registers per Block:           16384
Warp Size:                     32
Maximum Threads per Block:     512
Maximum Block Dimensions:      512, 512, 64
Maximum Grid Dimensions:       65535 x 65535 x 1
Maximum Memory Pitch:          2147483647B
Texture Alignment:             256B
Clock Rate:                    1296 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            No
ECC Enabled:                   No
Memory Clock Rate:             1107 MHz
Memory Bus Width:              512 bits
Max Threads Per SMP:           1024
Async Engines:                 1
Unified Addressing:            No
Initialization time:           856523 microseconds
Current free memory:           1015942912
Upload time (4MB):             8523 microseconds (4459 ms pinned)
Download time:                 15105 microseconds (4558 ms pinned)
Upload bandwidth:               492 MB/sec ( 940 MB/sec pinned)
Download bandwidth:             277 MB/sec ( 920 MB/sec pinned)
```

This tells you the CUDA driver version, the name and compute capability of the GPU (or GPUs, if you have more than one), the available memory, and so on.

Supporting Documentation and Examples

You may want to consult the OpenACC 1.0 specification, included with this release, for additional information. It is also available at the OpenACC website, http://www.openacc-standard.org. Simple examples appear in Chapter 3, Using OpenACC with the PGI Compilers.

An SDK is available at the OpenACC website. In future releases, it will be installed with the PGI compilers, at `/opt/pgi/linux86[-64]/2012/openacc/SDK`.

CHAPTER 2

Using OpenACC with the PGI Compilers

The OpenACC directives are enabled by adding the `-acc` or the `-ta=nvidia` flag to the PGI compiler command line. This release targets OpenACC to NVIDIA GPUs. See Chapter 3 for discussion about using OpenACC directives or the `-acc` flag with object files compiled with previous PGI releases using the PGI Accelerator directives. In particular, specifying either `-acc` or `-ta=nvidia` enables the OpenACC directives and the OpenACC runtime, as well as the PGI Accelerator Model directives.

This release does not fully implement the OpenACC 1.0 specification. Refer to Chapter 4, Implemented Features, for details about what features are included in this release, and what features are coming in updates over the next few months.

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 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 a positive integer value, 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 kernel  file=/user/guest/guide/a1.c function=vecaddgpu
line=6 device=0 grid=391 block=256
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_TIME` to a positive integer value, 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 like:

```
0 errors found

Accelerator Kernel Timing data
/home/mwolfe/OpenACC/guide/a1.c
  vecaddgpu
    5: region entered 1 time
        time(us): total=1051789 init=1047282 region=4507
                  kernels=53 data=3710
        w/o init: total=4507 max=4507 min=4507 avg=4507
        6: kernel launched 1 times
            grid: [391]  block: [256]
            time(us): total=53 max=53 min=53 avg=53
```

This tells you that the program entered one accelerator region and spent a total of about 1 second in that region. Most of that time was spent initializing the GPU (refer to 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
```

Troubleshooting Tips and Known Limitations). The program spent 3.7 milliseconds moving data between the host and GPU, and 53 microseconds executing the kernel. It also summarizes the time spent in each kernel generated from the OpenACC compute regions.

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 copyout(r[:n])
         Generating copyin(a[:n])
         Generating copyin(b[:n])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
      6, Loop is parallelizable
         Accelerator kernel generated
          6, #pragma acc loop gang, vector(256) /* blockIdx.x
threadIdx.x */
             CC 1.0 : 5 registers; 60 shared, 4 constant, 0 local
memory bytes; 100% occupancy
             CC 2.0 : 8 registers; 8 shared, 68 constant, 0 local
memory bytes; 100% occupancy
```

This tells you that the compiler generated two versions of the code, one for NVIDIA devices with compute capability 1.0 and higher, and one for devices with compute capability 2.0 and higher. It also gives the *schedule* used for the loop; in this case, the schedule is `gang, vector(256)`. This means the iterations of the loop are broken into vectors of 256, 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. 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 kernel generated
          6, #pragma acc loop seq
```

The compiler generated a sequential kernel, 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;
}
```

Fortran Examples

Vector Addition on the GPU

The simplest Fortran example of OpenACC is a 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 ACC_NOTIFY to a positive integer value, 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 kernel  file=/user/guest/guide/f1.f90 function=vecaddgpu
line=9 device=0 grid=391 block=256
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 391, and a thread block of size 256.

If you set the environment variable PGI_ACC_TIME to a positive integer value, 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 like:

```
0 errors found

Accelerator Kernel Timing data
/home/mwolfe/OpenACC/guide/f1.f90
  vecaddgpu
    8: region entered 1 time
        time(us): total=1016881 init=1013441 region=3440
                  kernels=53 data=2484
        w/o init: total=3440 max=3440 min=3440 avg=3440
        9: kernel launched 1 times
            grid: [391]  block: [256]
            time(us): total=53 max=53 min=53 avg=53
```

This tells you that the program entered one accelerator region and spent a total of about 1 second in that region. Most of that time was spent initializing the GPU (refer to the section 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
```

Troubleshooting Tips and Known Limitations). The program spent 2.4 milliseconds moving data between the host and GPU, and 53 microseconds executing the kernel. It also summarizes the time spent in each kernel generated from the OpenACC compute regions.

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 the previous program with the command `pgfortran -acc -fast -Minfo a1.c`, you get the following output:

```
vecaddgpu:
      8, Generating copyin(b(:n))
         Generating copyin(a(:n))
         Generating copyout(r(:n))
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
      9, Loop is parallelizable
         Accelerator kernel generated
          9, !$acc loop gang, vector(256) ! blockidx%x threadidx%x
             CC 1.0 : 4 registers; 72 shared, 4 constant, 0 local
memory bytes; 100% occupancy
             CC 2.0 : 10 registers; 8 shared, 80 constant, 0 local
memory bytes; 100% occupancy
```

This tells you that the compiler generated two versions of the code, one for NVIDIA devices with compute capability 1.0 and higher, and one for devices with compute capability 2.0 and higher. It also gives the *schedule* used for the loop; in this case, the schedule is `gang, vector(256)`. This means the iterations of the loop are broken into vectors of 256, 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 will 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
```

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
```

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 4, 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 takes about 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 compete.

This release does not allow a `present` data clause that specifies a C pointer which also appears in a lexically enclosing `data` construct, if the pointer value has changed.

In this release, the compiler does not automatically use the NVIDIA shared memory as a data cache.

CHAPTER 3

PGI Accelerator Model Interoperability

This chapter describes how the PGI OpenACC implementation interoperates with the PGI Accelerator model implementation, and with object files created with previous releases of the PGI compiler using the PGI Accelerator directives. PGI continues to support the PGI Accelerator model and directives as extensions to the OpenACC API. Where there were conflicts between the previous version of the PGI Accelerator model directives and OpenACC, the new PGI Accelerator model is now fully compatible with OpenACC.

OpenAcc New Features

OpenACC has added six important features that were not available in the PGI Accelerator model version 1.3.

- OpenACC has two types of compute constructs, the `acc parallel` construct and the `acc kernels` construct. The acc kernels construct is very similar to the PGI Accelerator `acc region` construct. The `acc parallel` construct is new to OpenACC. This release supports both the kernels and the parallel constructs.
- OpenACC has the `present`, `present_or_copy`, `present_or_copyin`, `present_or_copyout`, and `present_or_create` data clauses. These give functionality that is similar to the `reflected` data clause in the PGI Accelerator model, except the `present` clauses can be used for global data, and do not need an explicit interface. This release of the PGI OpenACC compilers supports the `present` clauses.

- OpenACC has support for asynchronous data movement between the host and GPU, and asynchronous computation on the GPU, using the `async` clause and the `wait` directive. This release of the PGI OpenACC compilers implements the `async` clause and asynchronous data movement and computation, and the `wait` directive. The `async` clause is supported on all relevant OpenACC directives, as well as PGI Accelerator directives.
- OpenACC defines three levels of parallelism: gang, worker and vector. The PGI Accelerator model version 1.3 defined two levels of parallelism: parallel and vector, where each level can have multiple dimensions. The OpenACC gang parallelism corresponds directly to the PGI Accelerator parallel level of parallelism. The OpenACC vector parallelism corresponds to the PGI Accelerator vector parallelism. The OpenACC worker parallelism is new. This release of the PGI OpenACC compilers supports worker parallelism.
- OpenACC supports an explicit `reduction` clause for inner loops. This release of the PGI OpenACC compilers supports the `reduction` clause on the parallel and kernels constructs, and on the loop constructs.
- OpenACC supports worker and vector parallelism for nontightly nested loops. The PGI Accelerator model version 1.3 only allowed vector parallelism for tightly nested outer loops. This release of the PGI OpenACC compilers supports worker and vector parallelism on nontightly nested loops.

There are other differences between OpenACC and the PGI Accelerator model, described in the following sections.

PGI Accelerator Features not available in the OpenACC

The PGI Accelerator Model has one feature that is not available in OpenACC.

- In Fortran, the PGI Accelerator model has `mirror` and `reflected` data clauses. The `mirror` data clause tells the compiler to allocate a device copy of the array whenever the host copy is allocated. For global arrays, the device copy is accessible in any subprogram where the host copy is accessible. The `reflected` data clause tells the compiler to pass the address of the device copy of an array as an argument when it passes the address of the host copy of the array. The functionality of both of these is similar to that of the `present` data clause in OpenACC. The advantage of `mirror` and `reflected` is that the compiler can

check at compile time whether the calling routine is passing an array with a device copy, or whether the global array has a device copy. The advantage of the `present` clause is that it doesn't require an explicit interface, and the same mechanism can be applied to dummy arguments and global arrays. PGI will continue to support the `mirror` and `reflected` clauses.

Changes from PGI Accelerator 1.3 to OpenACC

There were several incompatibilities in the PGI Accelerator model 1.3 and OpenACC. This release implements a new version of the PGI Accelerator model that is fully compatible with OpenACC. This change may affect how your program behaves, and may, in some cases, requires changes to your source program.

- The PGI Accelerator model version 1.3 allows for any rectangular subarray to be specified in a data clause, such as the interior of a matrix. OpenACC requires that data in a data clause be contiguous in memory. For instance, in the PGI Accelerator model, the following is legal:

```
subroutine sub(a,n,m)
 integer :: n, m
 real :: a(n,m)
 integer :: i,j
 !$acc region do copy( a(2:n-1,2:m-1) )
   do j = 2, m-1
     do i = 2, n-1
       a(i,j) = exp(a(i,j))
     enddo
   enddo
end subroutine
```

In OpenACC, the corresponding example would have to move a contiguous subarray, even though some of the elements moved are not used:

```
subroutine sub(a,n,m)
 integer :: n, m
 real :: a(n,m)
 integer :: i,j
 !$acc kernels loop copy( a(1:n,2:m-1) )
   do j = 2, m-1
     do i = 2, n-1
       a(i,j) = exp(a(i,j))
     enddo
   enddo
end subroutine
```

This new PGI Accelerator model uses the OpenACC interpretation that only device data must correspond to contiguous memory on the host. However, to promote portability and expressibility, if a noncontiguous subarray is specified in a data clause, as in the previous example, memory is allocated corresponding to the smallest contiguous region containing that subarray, and only the specified subarray is copied in either direction. This may change the behavior of your program, if the contiguous region is significantly larger than the subarray you specified.

- The new PGI Accelerator model allows for two dimensional dynamic C arrays, such as C `float**` arrays, to be moved to the accelerator. OpenACC does not currently allow this, because of the contiguous data requirement. In previous release, the two dimensional dynamic C array was linearized to a single long vector. The new PGI Accelerator model allocates and fills in a pointer vector in device memory, corresponding to the pointer vector on the host, as well as allocating and copying (if specified) the data array. This is an extension to OpenACC.
- In the PGI Accelerator model version 1.3, if no data region or data clause for an array is specified at an accelerator compute region, the compiler used `copyin`, `copyout` or `copy` for that array, depending on whether the array is read-only, written-only, or may be partially written or read before written. The new PGI Accelerator model implements the OpenACC default, which is to use `present_or_copyin`, `present_or_copyout` or `present_or_copy`.
- OpenACC has an explicit `reduction` clause for loops, the kernels constructs and the parallel construct. The PGI Accelerator model version 1.3 depended on the compiler to automatically detect reduction operations in the code. The new PGI Accelerator model allows for explicit `reduction` clauses, as well as automatically detected reduction operators.
- OpenACC has an explicit `cache` directive inside of loop. The PGI Accelerator model has a `cache` clause on the loop (`for` or `do`) construct, which is treated as a hint to the compiler. The new PGI Accelerator model allows for both.
- In C, the PGI Accelerator model version 1.3 used `x[lower:upper]` notation for subarrays in data clauses. The new PGI Accelerator model uses the OpenACC notation for subarrays in C, which is `x[lower:length]`. This may require changes to your program, if it uses subarrays in data clauses in C programs.

Mapping PGI Accelerator Features to OpenACC

Most of the PGI Accelerator Model features map directly to OpenACC features, with some important differences described in the next section.

- The general pragma and directive syntax are the same, with the same `acc` prefix.
- The PGI Accelerator `region` construct maps directly to the OpenACC `kernels` construct.
- The PGI Accelerator `data region` construct maps directly to the OpenACC `data` construct.
- The PGI Accelerator `copy`, `copyin` and `copyout` data clauses map almost directly to OpenACC, with some differences noted in the previous section relating to noncontiguous data regions.
- The PGI Accelerator `local` data clause maps to the OpenACC `create` data clause.
- The PGI Accelerator `deviceptr` data clause for C maps to the OpenACC `deviceptr` data clause for C.
- The PGI Accelerator `update device` and `update host` data clauses map directly to OpenACC.
- The PGI Accelerator `async` clause, `wait` directive and async API routines were defined to use an opaque handle. The OpenACC `async` clause, `wait` directive and async API routines use an integer expression. In this release, we have implemented the OpenACC specification for both OpenACC and the PGI Accelerator model.
- The PGI Accelerator `for` (C) and `do` (Fortran) directives map directly to the OpenACC `loop` directive.

Using the new PGI Accelerator Model with OpenACC

This release begins to implement the new PGI Accelerator model, which is fully compatible with OpenACC. If you specify the `-acc` or `-ta=nvidia` flag, the PGI Accelerator and OpenACC directives are enabled.

Object files created from PGI Accelerator model version 1.3 sources using older PGI releases may be linked with object files created from this release, by specifying either `-acc` or `-ta=nvidia` on the link command. However, arrays moved to the GPU in data

regions implemented in the PGI Accelerator routines will not be available with the `present` clause in OpenACC routines.

CHAPTER 4

Implemented Features

This chapter lists the OpenACC features available in this release, and what features will be implemented in upcoming PGI releases.

In This Release

The following OpenACC features are available in this release:

- The `kernels` construct and the `kernels loop` combined construct.
- The `parallel` construct and `parallel loop` combined construct.
- The `data` construct.
- The `cache` construct.
- The `if` clause on the `kernels, parallel,` and `data` constructs.
- The `async` clause on the `kernels` and `parallel` constructs.
- The `copy`, `copyin`, `copyout`, `create`, `present`, `present_or_copy`, `present_or_copyin`, `present_or_copyout` and `present_or_create` data clauses in C and Fortran.
- The `pcopy`, `pcopyin`, `pcopyout` and `pcreate` alternate spellings for `present`, `present_or_copy`, `present_or_copyin`, `present_or_copyout` and `present_or_create` data clauses.
- The `deviceptr` data clause for C.
- The `loop` construct, and the `seq`, `gang`, `worker`, `vector`, `independent`, `private`, `reduction` and `collapse` clauses for the loop construct.

- The `update` directive, and all its clauses.
- The `wait` directive.
- Implicit data regions.
- The `declare` directive, except for the `acc_resident` clause.
- The `openacc.h` header file for C.
- The `openacc_lib.h` header file and `openacc` module for Fortran.
- All the runtime API library routines.
- The environment variables `ACC_DEVICE_TYPE` and `ACC_DEVICE_NUM`.
- The `_OPENACC` preprocessor macro.

Defaults

In this release, the default `ACC_DEVICE_TYPE` is `acc_device_nvidia`, just as the `-acc` compiler option targets `-ta=nvidia` 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 PGI Accelerator Model and with the CUDA device numbering system. For more information, refer to the `pgaccelinfo` output on page 3. The device number can be changed using the environment variable or by a call to `acc_set_device_num`.

Known Limitations

Calls to acc_set_device_num when the device type is acc_device_nvidia may not have the desired effect once the CUDA context has been created.

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

In Future Releases

The following Open ACC features are not implemented in this release; they will appear in future releases:

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