PGI insider

Technical News from The Portland Group

PGI Home Page

June 2009

The PGI Accelerator Programming Model on NVIDIA GPUs Part 1

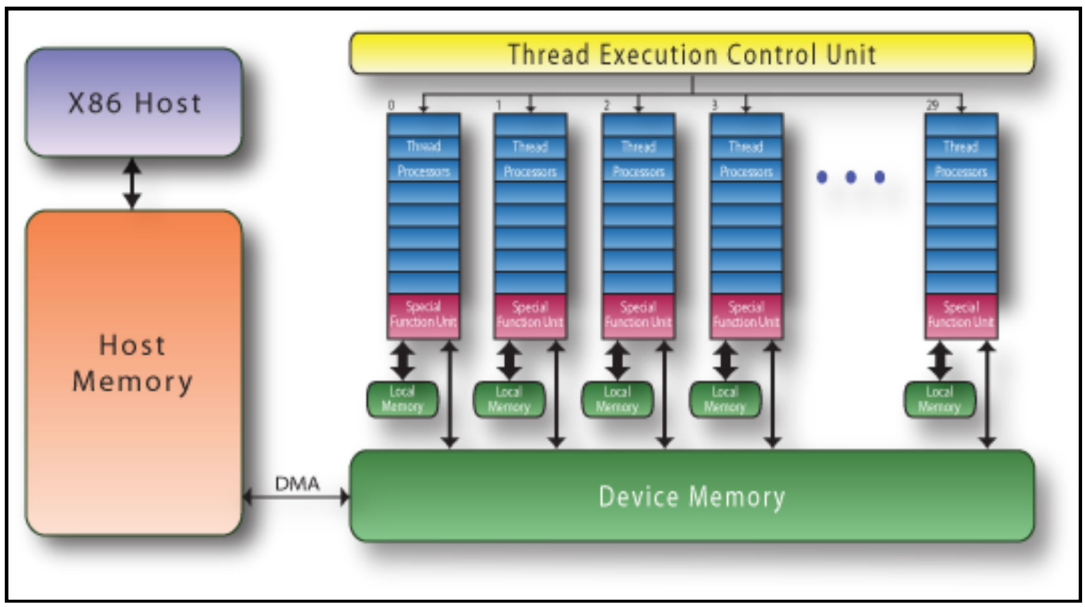
by Michael Wolfe, PGI Compiler Engineer

GPUs have a very high compute capacity, and recent designs have made them more programmable and useful for tasks other than just graphics. Research on using GPUs for general purpose computing has gone on for several years, but it was only when NVIDIA introduced the CUDA (Compute Unified Device Architecture) SDK, including a compiler with extensions to C, that GPU computing became useful without heroic effort. Yet, while CUDA is a big step towards GPU programming, it requires significant rewriting and restructuring of your program, and if you want to retain the option of running on an X64 host, you must maintain both the GPU and CPU program versions separately.

PGI is introducing the Accelerator Programming Model for Fortran and C with PGI Release 9.0. The Accelerator Programming Model uses directives and compiler analysis to compile natural Fortran and C for the GPU; this often allows you to maintain a single source version, since ignoring the directives will compile the same program for the X64 CPU. Note the model is called the *Accelerator Programming Model*, not the GPU Programming Model; the model is designed to be forward-looking as well, to accomodate other accelerators that may come in the future, preserving your software development investment.

GPU Architecture

Let's start by looking at accelerators, GPUs, and the NVIDIA GPU in particular, since it is the first target for our compiler. An accelerator is typically implemented as a *coprocessor* to the host; it has its own instruction set and usually (but not always) its own memory. To the hardware, the accelerator looks like another IO unit; it communicates with the CPU using IO commands and DMA memory transfers. To the software, the accelerator is another computer to which your program sends data and routines to execute. Many accelerators have been produced over the years; with current technology, an accelerator fits on a single chip, like a CPU. Today's accelerators include the Sony/Toshiba/IBM Cell Broadband Engine and GPUs. Here we focus on the NVIDIA family of GPUs; a picture of the relevant architectural features is shown below.



NVIDIA GPU Accelerator Block Diagram

The key features are the processors, the memory, and the interconnect. The NVIDIA GPUs have (currently) up to 30 *multiprocessors*; each multiprocessor has eight parallel *thread processors*. The thread processors run synchronously, meaning all eight thread processors run a copy of the same program, and actually execute the same instruction at the same time. Different multiprocessors run asynchronously, much like commodity multicore processors.

to another GPU thread, and continues executing that thread. In this way, the GPU exploits program parallelism to keep busy while the slow device memory is responding.

While the device memory has long latency, the interconnect between the memory and the GPU processors supports very high bandwidth. In contrast to a CPU, the memory can keep up with the demands of data-intensive programs; instead of suffering from cache stalls, the GPU can keep busy, as long as there is enough parallelism to keep the processors busy.

Programming

Current approaches to programming GPUs include NVIDIA's CUDA, AMD's Brook+, and the open standard language OpenCL. Over the past several years, there have been many success stories using CUDA to port programs to NVIDIA GPUs. The goal of OpenCL is to provide a portable mechanism to program different GPUs and other parallel systems. Is there need or room for another programming strategy?

The cost of programming using CUDA or OpenCL is the initial programming effort to convert your program into the host part and the accelerator part. Each routine to run on the accelerator must be extracted to a separate kernel function, and the host code must manage device memory allocation, data movement, and kernel invocation. The kernel itself may have to be carefully optimized for the GPU or accelerator, including unrolling loops and orchestrating device memory fetches and stores. While CUDA and OpenCL are much, much easier programming environments than what was available before, and both allow very detailed low-level optimization for the GPU, they are a long way from making it easy to program and experiment.

To address this, PGI has come up with our Accelerator Programming Model, implemented as a set of directives accepted by our Fortran and C compilers. Using these, you can more easily get started and experiment with porting programs to the NVIDIA GPUs, letting the compiler do much of the bookkeeping. The resulting program is more portable, and in fact can run unmodified on the CPU itself. In this first tutorial installment, we will show some initial programs to get you started, and explore some of the features of the model. As we will see in the next installment, this model doesn't make parallel programming *easy*, but it does reduce the cost of entry; we will also explore using the directives to tune performance.

Setting Up

You need the right hardware and software to start using the PGI Accelerator Model compilers. First, you need a 64-bit X64 system with a Linux distribution supported both by PGI and NVIDIA; these include recent RHEL, SLES, OpenSUSE, Fedora, and Ubuntu distributions. See the PGI release support page and the Download CUDA page on the NVIDIA web site for currently supported distributions. Your system needs a CUDA-enabled NVIDIA graphics or Tesla card; see the CUDA-Enable Products page on the NVIDIA web site for a list of appropriate cards. You need to install both the appropriate NVIDIA CUDA software and the PGI compilers. From NVIDIA, you'll want the latest CUDA driver, the CUDA toolkit, and CUDA SDK, all available from the Download CUDA page on the NVIDIA site.

Let's assume you've installed the CUDA software in /opt/cuda, and the PGI compilers under /opt/pgi; the PGI installation has two additional directory levels, corresponding to the target (32-bit linux86, or 64-bit linux86-64) and version (9.0 or 9.0-n, where n is the build number). Your installation will need to tell the PGI compilers where the CUDA software is installed. To do that, you'll want to create or modify the file "sitenvrc" in the /opt/pgi/linux86-64/9.0/bin directory to add the lines:

```
set NVDIR=/opt/cuda;
set NVOPEN64DIR=$NVDIR/open64/lib;
set CUDADIR=$NVDIR/bin;
set CUDALIB=$NVDIR/lib;
```

Remember the semicolons. You might also need to set your LD_LIBRARY_PATH environment variable to add the CUDA library directory; this is done with the following shell commands:

```
csh:   setenv LD_LIBRARY_PATH /opt/cuda/lib:"$LD_LIBRARY_PATH"
bash:  export LD_LIBRARY_PATH=/opt/cuda/lib:"$LD_LIBRARY_PATH"
```

Then you're ready to test your accelerator connection. Try running the PGI-supplied tool pgaccelinfo. If you've got everything set up properly, you should see output like:

```
Device Number:                  0
Device Name:                    Tesla C1060
Device Revision Number:         1.3
Global Memory Size:             4294705152
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:      16384
Maximum Block Dimensions:       512 x 512 x 64
Maximum Grid Dimensions:        65535 x 65535 x 1
Maximum Memory Pitch:           262144B
Texture Alignment               256B
Clock Rate:                     1296 MHz
```

This tells you that there is a single device, number zero; it's an NVIDIA Tesla C1060, it has compute capability 1.3, 1GB memory and 30 multiprocessors. You might have more than one GPU installed; perhaps you have a small GPU on the motherboard and a larger GPU or Tesla card in a PCI slot. The pgaccelinfo will give you information about each one it can find. On the other hand, if you see the message:

```
No accelerators found.
Try pgaccelinfo -v for more information
```

then you probably haven't got the right hardware or drivers installed.

If you have an older NVIDIA card with a "Device Revision Number" of 1.1 or 1.0, you will want to modify your compiler "siterc" file to set the default compute capability to 1.1 or 1.0. Go back to the /opt/pgi/linux86-64/9.0/bin directory and create or edit the file "siterc", and add the one line:

```
set COMPUTECAP=11;
```

or

```
set COMPUTECAP=10;
```

as appropriate.

Now you're ready to start your first program.

First Program

We're going to show several simple example programs; we encourage you to try each one yourself. In each case, we'll show the example in both C and Fortran; you can use whichever language you prefer. These examples are all available for download from the PGI web site at http://www.pgroup.com/lit/samples/pgi_accelerator_examples.tar.

We'll start with a very simple program; it will send a vector of floats to the GPU, double it, and bring the results back. In C, the whole program is:

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

int main( int argc, char* argv[] )
{
    int n;      /* size of the vector */
    float *restrict a;  /* the vector */
    float *restrict r;  /* the results */
    float *restrict e;  /* expected results */
    int i;
    if( argc > 1 )
        n = atoi( argv[1] );
    else
        n = 100000;
    if( n <= 0 ) n = 100000;
```

```
    a = (float*)malloc(n*sizeof(float));
    r = (float*)malloc(n*sizeof(float));
    e = (float*)malloc(n*sizeof(float));
    /* initialize */
    for( i = 0; i < n; ++i ) a[i] = (float)(i+1);

#pragma acc region
    {
        for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
    }
    /* compute on the host to compare */
        for( i = 0; i < n; ++i ) e[i] = a[i]*2.0f;
    /* check the results */
    for( i = 0; i < n; ++i )
        assert( r[i] == e[i] );
    printf( "%d iterations completed\n", n );
    return 0;
}
```

Note the `restrict` keyword in the declarations of the pointers; we'll see why shortly. Note also the explicit float constant `2.0f` instead of `2.0`. By default, C floating point constants are double precision. The expression `a[i]*2.0` is computed in double precision, as `(float)((double)a[i] * 2.0)`. To avoid this, use explicit float constants, or use the command line flag `-Mfcon`, which treats float constants as type float by default.

In Fortran, we would write:

```
program main
    integer :: n        ! size of the vector
    real,dimension(:),allocatable :: a  ! the vector
    real,dimension(:),allocatable :: r  ! the results
    real,dimension(:),allocatable :: e  ! expected results
    integer :: i
    character(10) :: arg1
    if( iargc() .gt. 0 )then
        call getarg( 1, arg1 )
        read(arg1,'(i10)') n
    else
        n = 100000
    endif
    if( n .le. 0 ) n = 100000
    allocate(a(n))
    allocate(r(n))
    allocate(e(n))
    do i = 1,n
        a(i) = i*2.0
    enddo
    !$acc region
        do i = 1,n
            r(i) = a(i) * 2.0
        enddo
    !$acc end region
        do i = 1,n
            e(i) = a(i) * 2.0
        enddo
    ! check the results
    do i = 1,n
        if( r(i) .ne. e(i) )then
            print *, i, r(i), e(i)
            stop 'error found'
        endif
    enddo
```

```
        print *, n, 'iterations completed'
    end program
```

In these programs, we enclosed the loop we want sent to the GPU in an accelerator region. In C, this is written as a #pragma acc region followed by a structured block in braces; in Fortran, we surround the region by !$acc region and !$acc end region directives. For this program, it's as simple as that.

Build these with the commands:

```
pgcc -o c1.exe c1.c -ta=nvidia -Minfo
```

or

```
pgfortran -o f1.exe f1.f90 -ta=nvidia -Minfo
```

Note the -ta and -Minfo flags. The -ta is the target accelerator flag; it tells the compiler to compile accelerator regions for the NVIDIA target accelerator. We'll show other options to this flag in later examples. The -Minfo flag enables informational messages from the compiler; we'll enable this on all our builds, and explain what the messages mean. You're going to want to understand these messages when you start to tune for performance.

If everything is installed and you have the accelerator licenses, you should see the following informational messages from pgcc:

```
main:
     22, Generating copyin(a[0:n-1])
         Generating copyout(r[0:n-1])
     24, Loop is parallelizable
         Accelerator kernel generated
             #pragma for parallel, vector(256)
```

or from pgfortran:

```
main:
     19, Generating copyin(a(1:n))
         Generating copyout(r(1:n))
     20, Loop is parallelizable
         Accelerator kernel generated
             !$acc do parallel, vector(256)
```

Let's explain a few of these messages. The first:

```
Generating copyin
```

tells you that the compiler determined that the array `a` is used only as input to the loop, so those `n` elements of `a` need to copied over from the CPU memory to the GPU device memory; this is a `copyin` to the device memory. Since they aren't modified, they don't need to be brought back. The second message

```
Generating copyout
```

tells you that the array `r` is assigned, but never read inside the loop; the values from the CPU memory don't need to be sent to the GPU, but the modified values need to be copied back. This is a `copyout` from the device memory. Below that is the message:

```
Loop is parallelizable
```

This tells you that the compiler analyzed the references in the loop and determined that all iterations could be executed in parallel. In the C program, we added the `restrict` keyword to the declarations of the pointers `a` and `r` to allow this; otherwise, the compiler couldn't safely determine that `a` and `r` pointed to different memory. The next message is the most key:

```
Accelerator kernel generated
```

This tells you that the compiler successfully converted the body of that loop to a kernel for the GPU. The kernel is the GPU function itself created by the compiler, that will be called by the program and executed in parallel on the GPU. We'll discuss the next message, and others that you'll see, in more detail in the next installment.

So now you're ready to run the program. Assuming you're on the machine with the GPU, just type the name of the

executable, c1.exe or f1.exe. If you get a message

```
libcuda.so not found, exiting
```

then you must not have installed the CUDA software in its default location, /usr/lib. You may have to set the environment variable LD_LIBRARY_PATH. What you should see is just the final output

```
100000 iterations completed
```

How do you know that anything executed on the GPU? You can set the environment variable ACC_NOTIFY to 1:

```
csh:   setenv ACC_NOTIFY 1
bash:  export ACC_NOTIFY=1
```

then run the program; it will then print out a line each time a GPU kernel is launched. In this case, you'll see something like:

```
launch kernel file=f1.f90 function=main line=22 grid=32 block=256
```

which tells you the file, function, and line number of the kernel, and the CUDA grid and thread block dimensions. You probably don't want to leave this set for all your programs, but it's instructive and useful during program development and testing.

Second Program

Our first program was pretty trivial, just enough to get a test run. Let's take on a slightly more interesting program, one that has more computational intensity on each iteration. Again, I'll show the whole program in C and in Fortran. In C, the program is:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/time.h>
#include <math.h>
#include <accel.h>
#include <accelmath.h>

int main( int argc, char* argv[] )
{
    int n;      /* size of the vector */
    float *restrict a;  /* the vector */
    float *restrict r;  /* the results */
    float *restrict e;  /* expected results */
    float s, c;
    struct timeval t1, t2, t3;
    long cgpu, chost;
    int i;
    if( argc > 1 )
        n = atoi( argv[1] );
    else
        n = 100000;
    if( n <= 0 ) n = 100000;

    a = (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) * 2.0f;
    /*acc_init( acc_device_nvidia );*/

    gettimeofday( &t1, NULL );
```

```
        #pragma acc region
        {
            for( i = 0; i < n; ++i ){
                s = sinf(a[i]);
                c = cosf(a[i]);
                r[i] = s*s + c*c;
            }
        }
        gettimeofday( &t2, NULL );
        cgpu = (t2.tv_sec - t1.tv_sec)*1000000 + (t2.tv_usec - t1.tv_usec);
            for( i = 0; i < n; ++i ){
                s = sinf(a[i]);
                c = cosf(a[i]);
                e[i] = s*s + c*c;
            }
        gettimeofday( &t3, NULL );
        chost = (t3.tv_sec - t2.tv_sec)*1000000 + (t3.tv_usec - t2.tv_usec);
        /* check the results */
        for( i = 0; i < n; ++i )
            assert( fabsf(r[i] - e[i]) < 0.000001f );
        printf( "%13d iterations completed\n", n );
        printf( "%13ld microseconds on GPU\n", cgpu );
        printf( "%13ld microseconds on host\n", chost );
        return 0;
    }
```

It reads the first command line argument as the number of elements to compute, allocates arrays, runs a kernel to compute floating point sine and cosine (note sinf and cosf function names here), and compares to the host. Some details to note:

- I have a call to acc_init() commented out; you'll uncomment that shortly.
- I have calls to gettimeofday() to measure wall clock time on the GPU and host loops.
- I don't compare for equality, I compare against a tolerance. We'll discuss that as well.

The same program in Fortran is

```
    program main
        use accel_lib
        integer :: n        ! size of the vector
        real,dimension(:),allocatable :: a  ! the vector
        real,dimension(:),allocatable :: r  ! the results
        real,dimension(:),allocatable :: e  ! expected results
        integer :: i
        integer :: c0, c1, c2, c3, cgpu, chost
        character(10) :: arg1
        if( iargc() .gt. 0 )then
            call getarg( 1, arg1 )
            read(arg1,'(i10)') n
        else
            n = 100000
        endif
        if( n .le. 0 ) n = 100000
        allocate(a(n))
        allocate(r(n))
        allocate(e(n))
        do i = 1,n
            a(i) = i*2.0
        enddo
        !call acc_init( acc_device_nvidia )
        call system_clock( count=c1 )
        !$acc region
            do i = 1,n
```

```
                r(i) = sin(a(i)) ** 2 + cos(a(i)) ** 2
            enddo
        !$acc end region
        call system_clock( count=c2 )
        cgpu = c2 - c1
            do i = 1,n
                e(i) = sin(a(i)) ** 2 + cos(a(i)) ** 2
            enddo
        call system_clock( count=c3 )
        chost = c3 - c2
        ! check the results
        do i = 1,n
            if( abs(r(i) - e(i)) .gt. 0.000001 )then
                print *, i, r(i), e(i)
            endif
        enddo
        print *, n, ' iterations completed'
        print *, cgpu, ' microseconds on GPU'
        print *, chost, ' microseconds on host'
    end program
```

Here, we use system_clock to read the real time clock. Note the lib3f calls to `iargc` and `getarg` to get command line arguments. You can also replace these with calls to the more recent `command_argument_count()` and `get_command_argument()` routines.

Now let's build and run the program; you'll compare the speed of your GPU to the speed of the host. Build as you did before, and you should see messages much like you did before. You can view just the accelerator messages by replacing `-Minfo` by `-Minfo=accel` on the compile line.

The first time you run this program, you'll see output something like:

```
 100000 iterations completed
1016510 microseconds on GPU
   2359 microseconds on host
```

So what's this? A second on the GPU? Only 2.3 milliseconds on the host? What's the deal?

Let's explore this a little. If I enclose the program from the first call to the timer to the last print statement in another loop that iterates three times, I'll see something more like the following:

```
 100000 iterations completed
1262463 microseconds on GPU
   2342 microseconds on host
 100000 iterations completed
   1166 microseconds on GPU
   1884 microseconds on host
 100000 iterations completed
   1145 microseconds on GPU
   1901 microseconds on host
```

The time on the GPU is very long for the first iteration, then is much faster after that. The reason is the overhead of connecting to the GPU; depending on your system; it can take 1 to 1.5 seconds to make that initial connection, the first time the first kernel executes. You can see this more clearly by building the original program with `-ta=nvidia,time`. This includes a profiling library that collects the time spent in GPU initialization, data movement, and kernel execution. If we execute the program built that way, we'll get additional profile information:

```
 100000 iterations completed
1182693 microseconds on GPU
   2337 microseconds on host

Accelerator Kernel Timing data
c2.c
  main
    32: region entered 1 times
        time(us): total=1182682 init=1180869 region=1813
                  kernels=170 data=1643
        w/o init: total=1813 max=1813 min=1813 avg=1813
        34: kernel launched 1 times
```

```
    time(us): total=170 max=170 min=170 avg=170
```

The timing data tells us that the accelerator region at line 32 was entered once and took a total of 1.182 seconds. Of that, 1.180 was spent in initialization. The actual execution time was 1.8 milliseconds. Of that time, 1.6 milliseconds was spent moving data back and forth (copying the `a` and `r` arrays to and from the GPU), and only 170 microseconds was spent executing the kernel.

So, let's take the initialization out of the timing code altogether. Uncomment the call to acc_init() in your program, rebuild and then run the program. You should see output more like:

```
100000 iterations completed
  1829 microseconds on GPU
  2341 microseconds on host
```

The GPU time still includes the overhead of moving data between the GPU and host. Your times may differ, even substantially, depending on the GPU you have installed (particularly the Number of Multiprocessors reported by pgaccelinfo) and the host processor. These runs were made on a 2.4GHz AMD Athlon 64.

To see some more interesting performance numbers, try increasing the number of loop iterations. The program defaults to 100000; increase this to 1000000. On my machine, I see

```
1000000 iterations completed
   9868 microseconds on GPU
  54712 microseconds on host
```

Note the GPU time increases by about a factor of 10, as you would expect; the host time increases by quite a bit more. That's because the host is sensitive to cache locality. The GPU has a very high bandwidth device memory; it uses the extra parallelism that comes from the 1,000,000 parallel iterations to tolerate the long memory latency, so there's no performance cliff.

So, let's go back and look at the reason for the tolerance test, instead of an equality test, for correctness. The fact is that the GPU doesn't compute to exactly the same precision as the host. In particular, some transcendentals and trigonometric functions may be different in the low-order bit. You, the programmer, have to be aware of the potential for these differences, and if they are not acceptable, you may need to wait until the GPUs implement full host equivalence. Before the adoption of the IEEE floating point arithmetic standard, every computer used a different floating point format and delivered different precision, so this is not a new problem, just a new manifestation.

Your next assignment is to convert this second program to double precision; for C programmers, remember to change the `sinf` and `cosf` calls. You might compare the results to find the maximum difference. We're computing sin^2 + cos^2, which, if I remember my high school geometry, should equal 1.0 for all angles. So, you can compare the GPU and host computed values against the actual correct answer as well.

Third Program

Here, we'll explore writing a slightly more complex program, and try some other options, such as building it to run on either the GPU, or on the host if you don't have a GPU installed. We'll look at a simple Jacobi relaxation on a two-dimensional rectangular mesh. In C, the relaxation routine we'll use is:

```
typedef float *restrict *restrict MAT;

void
smooth( MAT a, MAT b, float w0, float w1, float w2,
        int n, int m, int niters )
{
  int i, j, iter;
  #pragma acc region
  {
    for( iter = 1; iter < niters; ++iter ){
      for( i = 1; i < n-1; ++i )
        for( j = 1; j < m-1; ++j )
          a[i][j] = w0 * b[i][j] +
              w1*(b[i-1][j]+b[i+1][j]+b[i][j-1]+b[i][j+1]) +
              w2*(b[i-1][j-1]+b[i-1][j+1]+b[i+1][j-1]+b[i+1][j+1]);
      for( i = 1; i < n-1; ++i )
        for( j = 1; j < m-1; ++j )
          b[i][j] = a[i][j];
    }
  }
}
```

Again, note the use of the restrict keyword on the pointers. In Fortran, the routine looks similar:

```
       subroutine smooth( a, b, w0, w1, w2, n, m, niters )
         real, dimension(:,:) :: a,b
         real :: w0, w1, w2
         integer :: n, m, niters
         integer :: i, j, iter
         !$acc region
           do iter = 1,niters
             do i = 2,n-1
               do j = 2,m-1
                 a(i,j) = w0 * b(i,j) + &
                     w1*(b(i-1,j)+b(i,j-1)+b(i+1,j)+b(i,j+1)) + &
                     w2*(b(i-1,j-1)+b(i-1,j+1)+b(i+1,j-1)+b(i+1,j+1))
               enddo
             enddo
             do i = 2,n-1
               do j = 2,m-1
                 b(i,j) = a(i,j)
               enddo
             enddo
           enddo
         !$acc end region
       end subroutine
```

We can build these routines as before, and we'll get a set of messages as before. This particular implementation executes a fixed number of iterations. Note the compiler message about the `iter` loop; it gets scheduled on the host, and the inner loops get turned into two GPU kernels.

But what we want here is a program that will run on the GPU when it's available, or on the host when it's not. To do that, we build with the -ta=nvidia,host option. This generates two versions of this routine, one that runs on the host and one on the GPU, using the PGI Unified Binary technology. At run time, the program will determine whether there is a GPU attached and run that version if there is, or run the host version if there is not. You should see compiler messages like:

```
smooth:
      3, PGI Unified Binary version for -tp=k8-64e -ta=host
     10, Loop interchange produces reordered loop nest: 11,10
         ...
smooth:
      3, PGI Unified Binary version for -tp=k8-64e -ta=nvidia
      8, Generating copyout(a(2:n-1,2:m-1))
         Generating copyin(b(1:n,1:m))
         Generating copyout(b(2:n-1,2:m-1))
         ...
```

where the printed -tp value depends on the host on which you are running. Now you should be able to run this on the machine with the GPU and see the GPU performance, and then move the same binary to a machine with no GPU, where the host (-ta=host) copy will run.

By default, the runtime system will use the GPU version if the GPU is available, and will use the host version if it is not. You can manually select the host version two ways. One way is to set the environment variable ACC_DEVICE before running the program:

```
       csh:   setenv ACC_DEVICE host
       bash:  export ACC_DEVICE=host
```

You can go back to the default by unsetting this variable. Alternatively, the program can select the device by calling the `acc_set_device` routine:

```
       C:      #include "accel.h"
```

```
        ...
        acc_set_device( acc_device_host )
```

```
Fortran:
        use accel_lib
        integer :: n      ! size of the vector
        ...
        call acc_set_device( acc_device_host )
```

Summary

This installment introduced the PGI Accelerator model for NVIDIA GPUs, and presented three simple programs in C and Fortran. We looked at some issues you may run into, particularly C, with float vs. double, and unrestricted pointers. We presented the target accelerator flag, using the simple accelerator profile library with -ta=nvidia,time, and using -ta=nvidia,host to generate a unified host+GPU binary.

We hope you have a chance to try our simple examples and can start putting some simple programs on the GPU. The next installment will look at performance tuning, in particular looking at data movement between the host and the GPU, and at the loop schedules. It will also discuss the compiler messages in more detail.