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](image-url)
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.

The GPU has its own memory, usually called device memory; this can range up to 4GB today. As with CPUs, access time to the memory is quite slow. CPUs use caches to try to reduce the effect of the long memory latency, by caching recently accessed data in the hopes that the future accesses will hit in the cache. Caches can be quite effective, but they don't solve the basic memory bandwidth problem. GPU programs typically require streaming access to large data sets that would overflow the size of a reasonable cache. To solve this problem, GPUs use multithreading. When the GPU processor issues an access to the device memory, that GPU thread goes to sleep until the memory returns the value. In the meantime, the GPU processor switches over very quickly, in hardware, 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:

```c
#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:

```fortran
program main
    integer :: n         ! size of the vector
```

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 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:

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

or

```bash
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:
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 be 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:

```c
#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;
    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);
}
```

```fortran
program main
  implicit none
  integer n
  real, allocatable :: a(:), r(:), e(:)
  integer i
  call alloc(a, n, real)
  call alloc(r, n, real)
  call alloc(e, n, real)
  do i = 1, n
    a(i) = (i+1) * 2.0f
  end do
  !acc_init( acc_device_nvidia )
  call gettimeofday(t1)
  do i = 1, n
    s = sin(a(i))
    c = cos(a(i))
    r(i) = s*s + c*c
  end do
  call gettimeofday(t2)
  cgpu = (t2.tv_sec - t1.tv_sec)*1000000 + (t2.tv_usec - t1.tv_usec)
end program main
```
```c
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

```fortran
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
```
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
```
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:

```plaintext
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:

```plaintext
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 transcendental 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:

```c
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] +  
```
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
```
```
    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.
The PGI Accelerator Programming Model on NVIDIA GPUs
Part 2 Performance Tuning

by Michael Wolfe, PGI Compiler Engineer

The first installment of this series introduced the PGI Accelerator Programming Model, showing three simplified programs in C and in Fortran, and presenting a few details of building and running a program on the GPU. Here we discuss some issues affecting performance and how to recognize and address them. We'll discuss the four most important performance issues: writing an appropriately parallel algorithm, tuning the data movement between the host and the accelerator, tuning memory loads and stores on the accelerator, and tuning the loop schedule. For all of these, we'll need to understand how to interpret the compiler messages in order to tune the behavior. We'll also want to be able to measure the performance, to see how our tuning efforts pay off. This column will give you performance tuning guidelines, as well as sharpen your performance analysis and measurement skills.

The PGI Accelerator Fortran and C99 compilers available in PGI Release 9.0 target the NVIDIA CUDA-enabled GPUs and Tesla cards, and the tuning guidelines given here apply to those products. We'll use some of the enhanced features in the PGI 9.0 - 3 build, which should be available by the time you read this. We'll discuss tuning for other accelerator targets in future columns, as they are supported. The example programs here were run on our test system, a 2.67GHz, quad-core Intel Nehalem processor running Linux (RHEL 5.3), with an attached NVIDIA Tesla C1060 at 1.3GHz with 4GB memory. All the example programs are available from the PGI web site at www.pgroup.com/lit/samples/pginsider_v1n2a1_examples.tar.

Summary

Let's start with an executive summary. The most important part of a successful accelerated program is an appropriate parallel algorithm. NVIDIA GPUs are designed as throughput engines, to compute massive amounts of video data in parallel; they are designed to take advantage of the massive parallelism in several ways. First, GPUs can compute on different parts of the data set in parallel on different cores. Second, GPUs are designed to be more efficient by executing threads in groups, which NVIDIA calls warps; you can loosely think of a warp as a group of threads that execute in SIMD or vector mode, though that's not really how it's implemented. Third, the GPU can use the extra parallelism to keep the cores busy even when threads are waiting on some long latency operation, such as memory. With enough parallelism, GPU cores don't stall, unlike a CPU waiting for a cache miss.

In a PGI Accelerator Model program, parallelism is expressed in parallel loops. A high performance program will have one or more parallel loops, with lots (and lots) of iterations. If there is a single parallel loop with thousands of iterations, it can be successfully and effectively converted to an efficient GPU kernel. On the other hand, a single parallel loop with only a hundred or so iterations does not contain enough parallelism to keep the accelerator busy; nested parallel loops are necessary to generate enough parallelism with small loop limits.

Once you have an appropriate algorithm, you may want to tune the data movement between the host and the accelerator. The PGI compilers will automatically determine which arrays or array sections need to be sent to the accelerator from the host, and which modified arrays or array sections need to come back. However, a savvy programmer can tune the compiler analysis in two ways. First, the compiler tries to minimize the data traffic by sending the smallest array sections in either direction; this will often send a noncontiguous section of an array, such as the interior of a rectangular matrix. Because of the way the data transfers are done, it's usually much more efficient to send one large contiguous section; even if it means sending more data bytes, one transfer may map to a single DMA operation and be faster overall. A programmer can direct the compiler to do this with a data clause on the accelerator region directive. Second, the compiler (currently) will always bring data that is modified on the GPU back to the host. If that data isn't used after the accelerator region, the programmer can direct the compiler not to bring the data back, again with a data clause on the region directive. We'll see examples of these along with their effectiveness.

Then we must look at the performance of the parallel loops, or the kernel(s). First, look at the messages about non-stride-1 array accesses. NVIDIA GPUs access the device memory as 64-byte memory lines or
superwords. The compiler tries to optimize the loop schedule to maximize the number of stride-1 array accesses, and it gives a message when there are one or more non-stride-1 accesses. You may be able to force stride-1 accesses by modifying the loop schedule, or you may want to change the array layout, transposing array dimensions, to get better memory performance.

Finally, the loop schedule itself plays a large part in the final performance. The loop schedule is the mapping of loop parallelism onto the hardware parallelism. When programming directly in CUDA C, you choose how to map your computation onto a grid of thread blocks, the size of each dimension of the grid and thread blocks, and how to use the block and thread indices to compute array and loop indices; we call this the scheduling problem. The current state of the art method to solve the scheduling problem is to try many variations of your program, with different grid and thread block sizes and index mappings, to find the most efficient one. A compiler, of course, can't do such a search, since it can't run the program. The PGI compilers have an optimization procedure, which we're constantly tuning, but there are many cases where a programmer can get better performance by manually tuning the schedule.

Your GPU

First, you should know what kind of NVIDIA GPU you are targeting. NVIDIA has a broad family of CUDA-enabled GPUs, which differ in many respects. The high end cards, like our Tesla, have 30 multiprocessors and 4GB device memory; they have a compute capability of 1.3, which means they have hardware to support double-precision operations and better memory coalescing. Lower end cards will have fewer multiprocessors, less device memory, and a lower compute capability. You can see the features of your card using the pgaccelinfo command that comes with the PGI compilers. The output will look something like:

<table>
<thead>
<tr>
<th>Device Number:</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Name:</td>
<td>Tesla C1060</td>
</tr>
<tr>
<td>Device Revision Number:</td>
<td>1.3</td>
</tr>
<tr>
<td>Global Memory Size:</td>
<td>4294705152</td>
</tr>
<tr>
<td>Number of Multiprocessors:</td>
<td>30</td>
</tr>
<tr>
<td>Number of Cores:</td>
<td>240</td>
</tr>
</tbody>
</table>

The Device Revision Number is the compute capability, telling us that this card has double-precision support. If you have a revision 1.2 or lower, you can only run single-precision floating point on the card. There are also improvements in the way the GPU accesses memory with higher revisions, as we'll explain later. You can expect that a card with more multiprocessors will run your compute kernels that much faster than a card with fewer.

Appropriate Algorithm

To successfully run your program on a GPU, you must have an appropriately parallel algorithm. It must have lots of data parallelism, meaning that the program does the same operation over lots of data. In your program, this means a parallel loop, or even better, nested parallel loops. Let's take for example a four-point difference equation, here written in C:

```c
for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
        a[i][j] = w0 * a[i][j] +
                  w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

This nested loop has no parallelism; the \( j \) loop carries a dependence because of the assignment to \( a[i][j] \) in one \( j \) iteration, and the use of that value in the next \( j \) iteration when fetching \( a[i][j-1] \). Similarly, the \( i \) loop carries a dependence because of the assignment to \( a[i][j] \) in one \( i \) iteration, and the use of that value when fetching \( a[i-1][j] \) for the next \( i \) iteration. If we put this loop in an accelerator region and build with `pgcc -ta=nvidia -Minfo`, we'll see messages to that effect:

```
test:
32, No parallel kernels found, accelerator region ignored
34, Loop carried dependence of 'a' prevents parallelization
      Loop carried backward dependence of 'a' prevents vectorization
35, Loop carried dependence of 'a' prevents parallelization
```
Loop carried backward dependence of 'a' prevents vectorization

At line 35 (the j loop), the messages tell us there are loop carried dependences for the array a in the inner loop; there are two messages because the dependence described above from the left hand side to the right hand side expression is treated as a lexically backward dependence, since the left hand side is assigned after the right hand side has been evaluated. There is also a loop carried write-after-read dependence from the use of a[i][j+1] to its reassignment in the next j iteration. Similarly, line 34 (the i loop) has loop carried dependences as well. Since there are no parallel loops, the compiler fails to generate any kernels, and ignores the accelerator region entirely.

However, we may be able to recast this as a parallel loop by using a Jacobi method, instead of the Gauss-Seidel method:

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    b[i][j] = w0*a[i][j] +
              w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    a[i][j] = b[i][j];
```

In this formulation, both loops around the first (b) assignment are parallel, because no values being assigned are used in the right hand side. Similarly, both loops around the second (a) assignment are parallel, for the same reason.

These difference equations are used in iterative computations; the Jacobi method is known to require more iterations to converge, but you may be able to improve your total performance even so, by computing each iteration in parallel. To generalize, taking the best sequential program and simply parallelizing it may not produce the best parallel program. You need to balance any overhead of a parallel program against the speedup you get by running in parallel. On today's GPUs, there's a lot of parallelism available, so it can absorb a lot of overhead.

The loops that will run well on a GPU have many parallel iterations. Nested loops will have two or more levels of parallelism. Note that the inner loop limits must be invariant in the outer loops; the GPU structures for nested parallelism require a rectangular domain. For efficient memory access, the arrays should have stride-1 access in one of the loops; the compiler tries to optimize for stride-1 accesses by reordering the nested loops, but it won't rearrange the data in memory. The body of the loop can be as small as a single statement, or it can be quite large, including nested internal loops; more on this later. Today's GPUs don't support procedure calls, so any functions called in the loop body will have to be inlined, manually by the programmer or automatically by the compiler.

**Hindrances to GPU Parallelism**

As you experiment with and port programs to the GPU, you will run into many cases where the compiler fails to generate parallel code. The compiler will produce informational messages (with `-Minfo`) that you'll want to know how to read. Here, we'll explain the most common messages you'll run into, and how to work around them.

```
for( i = 0; i < n-1; ++i )
  b[i] = a[i] + a[i+1];
```

This message can occur for several different reasons. One common idiom we use in programs is to save a value in a scalar for later reuse. For instance, instead of writing

```
  x = a[0];
  for( i = 0; i < n-1; ++i ){
    y = a[i+1];
    b[i] = x + y;
  }
```

we might try to save one array fetch in the loop by using a scalar:
Such micro-optimization is particularly common in C programs, where it's even more likely that the programmer uses pointer arithmetic instead of array accesses. However, this second program has a loop-carried dependence for the scalar $x$. The value of $x$ is assigned on one iteration and used on the next iteration, so the iterations are no longer independent. The original loop was completely parallel, so the problem is not the algorithm, it's just the program.

Another example comes from conditional assignments to a scalar.

```c
for( i = 0; i < n; ++i ){
    if( a[i] < 0 ) x = b[i]*w;
    c[i] = x*a[i];
}
```

In this example, the scalar $x$ is only assigned on some of the iterations; for those iterations where it isn't assigned, the value must come from some previous iteration, again hindering parallelism. This can even occur when the use is also conditional:

```c
for( i = 0; i < n; ++i ){
    if( a[i] < 0 ) x = b[i]*w;
    c[i] = x*a[i];
}
```

In this example, the assignment and use are both conditional. The compiler, however, isn't clever enough to determine that the use only occurs under the same condition as the assignment, and so the value from the previous iteration is never needed. Rewriting this so the assignment and use are under the same condition removes the problem:

```c
for( i = 0; i < n; ++i ){
    if( a[i] < 0 )
        x = b[i]*w;
    c[i] = x*a[i];
    d[i] = a[i]+1;
}
```

The current release of the compiler will give the same message for reduction loops as well, such as:

```c
sum = 0.0;
for( i = 0; i < n; ++i ){
    sum += a[i] * b[i];
}
```

There are well-known methods to parallelize reductions, and future releases of the PGI Accelerator compilers will implement these automatically.

**Scalar last value needed after loop for 'x'**

This message occurs when a scalar assigned in the loop is *live* after the loop; that is, its value is used (or the compiler thinks it might be used) after the loop.

```c
for( i = 0; i < n; ++i ){
    x = b[i]*w;
    if( x != 0 ) c[i] = x*a[i];
}
if( x > 0 ){
    ...
}
The loop itself may be parallel, but to do so the compiler has to privatize the scalars (make a private copy for each iteration), and the values are lost after the loop. Usually in these cases, the last value isn't really needed; perhaps the variable is a global variable (C `extern`), or there's some other use of the variable where the compiler can't prove that the value from the loop is never used again. Renaming the variable inside the loop so it's distinct from other variables in the program will solve this problem, or inserting a reassignment after the loop so the compiler knows that the value from the loop never reaches any uses outside is just as good.

Loop carried dependence of 'a' prevents parallelization

This message comes out when there is a dependence due to an array reference. For C programs, we'll assume that all array pointers are declared with the `restrict` modifier, or the program is compiled with the `-Msafeptr` option; otherwise, the compiler can't tell whether any two array pointers might conflict. It might be a recurrence relation, or a simpler dependence from one iteration to another, or there might be a complex array subscript or an array that isn't indexed by the loop index. A real recurrence relation is shown below, where the value assigned in one iteration is used to compute the next value:

```c
for( i = 1; i < n; ++i )
    a[i] = a[i-1]*b[i] + c[i];
```

In this case, there is a real dependence; there are some very complex methods to directly parallelize loops like this, but they are relatively inefficient and can suffer from numerical accuracy problems. There are no easy solutions for these cases.

A simpler case is the following:

```c
for( i = 1; i < n; ++i )
    a[i] = b[i] + c[i];
    d[i] = a[i-1] + 2*a[i];
}
```

Here, there is a loop-carried dependence because of the assignment to `a[i]` and its use in the next iteration as `a[i-1]`. One way to work around the problem in this case is to split the loop into two loops:

```c
for( i = 1; i < n; ++i )
    d[i] = a[i-1] + 2*a[i];
```

Each of the two loops is now parallel; there is some cost since the `d` assignment has to fetch both operands from memory, but parallelism will often trump the small inefficiency.

When index arrays are used, the compiler has to assume there might be conflicts between iterations:

```c
for( i = 0; i < n; ++i )
    a[ndx[i]] += b[i];
```

Sometimes, the operation can be recast so the index array is used on the right hand side:

```c
for( i = 0; i < n; ++i )
    a[i] += b[rndx[i]];  
```

Since the array `b` isn't assigned in the loop, there are no dependence conflicts.

Finally, we might have a case with a one-dimensional array in a doubly nested loop:

```c
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++i )
        a[i] = 0.0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++i )
        if( b[i][j] > 0.0 ) a[i] += b[i][j];
```
if( b[i][j] > 0.0 ) a[i] += b[i][j];
}
}

In this case, the conditional assignment to a in the j loop can't be done in parallel; the left hand side doesn't use the j index at all, so each iteration is trying to assign the same element. However, the outer i loop can be successfully parallelized.

Parallelization would require privatization of array 'a[0:n-1]'

This message is produced when the compiler notices that the array is a candidate for adding to a private clause for the loop. It's safe to do this if all uses of that array come from assignments in that loop, and the array isn't used after the loop. As noted above, the compiler automatically privatizes scalars, but currently doesn't automatically privatize arrays. An example is:

    for( i = 0; i < n; ++i ){
        for( j = 0; j < n; ++j ){
            a[j] = b[i][j] * w + c[i]*d[j];
            if( a[j] > 0 ) p[i] -= a[j];
        }
    }

Here, the uses of a[j] come from the assignment in the loop. The j loop can't be parallelized because of the conditional assignment to p[i], which doesn't use the j index. However, if the array a isn't used after the loop, it can be privatized, allowing the compiler to parallelize the outer i loop. We can do this by adding the loop directive:

    #pragma acc for private(a[0:n-1])

In the accelerator model, the compiler converts the body of some loop (or loops) into a kernel or kernels. Sometimes, the way the program is written prevents the obvious mapping. For instance, a conditional that branches around an inner loop will prevent nested loops from both executing in parallel.

    for( i = 0; i < n; ++i )
        if( c[i] > 0 )
            for( j = 0; j < n; ++j )
                a[i][j] += b[i][j] * c[i];

Here, the compiler can either turn the outer i loop body into a kernel, running the i loop in parallel, or turn the inner j loop body into a kernel, running the j loop in parallel. However, it can't run both loops in parallel because the inner loop is only conditionally executed. Rewriting this as:

    for( i = 0; i < n; ++i )
        if( c[i] > 0 )
            for( j = 0; j < n; ++j )
                a[i][j] += b[i][j] * c[i];

Allows both loops to execute in parallel.

**Accelerator and Compiler Restrictions**

There are a number of restrictions on what is allowed in an accelerated loop for the NVIDIA GPUs; some of these are limitations of the device, some are limitations imposed by the compiler. You might see messages such as:

- Accelerator restriction: unsupported operation
- Accelerator restriction: function/procedure calls are not supported
- Accelerator restriction: struct/member references are not yet supported
- Accelerator restriction: datatype not supported
Some of these restrictions will be lifted in future compiler releases; until then, that part of the program will have to be left on the host, or rewritten to avoid the restriction.

Accelerator compiler license not found, accelerator code generation disabled.

If you see this message, then it's likely that you either haven't received or haven't installed the license for the accelerator features, or that perhaps it's expired. Either install your license, or contact PGI Support for an updated license.

**Parallelizable Loops**

The compiler notes which loops are potentially parallelizable with this message. This doesn't mean the loop will be executed in parallel, just that there are no dependences that would prevent it. We will show below why the compiler might choose not to run a parallelizable loop in parallel.

**Accelerator kernel generated**

This message appears with the line number of the top of some loop; it tells you that the body of that loop will be the kernel, and that the enclosing loops will be executed in parallel according to the loop schedule described just after this message. We'll describe the loop schedules below.

**Performance Analysis**

Once you have an appropriately parallel program that successfully generates accelerator kernels, you can start to look at performance. Here, we'll look at three ways to inspect the performance. We'll look at the compiler messages below, when we start looking at specific performance issues. We'll run the program with the profile library. And finally, we'll run the program under control of NVIDIA's cudaproc tool.

PGI Release 9.0 includes a simple accelerator profile feature, enabled by adding an option to the target accelerator command line flag. By linking with the -ta=nvidia,time flag, the profile library is linked into the program. When the program is run, performance is collected for accelerator regions and kernels. Specifically, it collects the elapsed time spent in the accelerator region and in each kernel in the region, and separates out compute time from host-GPU communication time. It also shows how much time is spent initializing the device; we'll show a little trick to reduce that as well.

After building with -ta=nvidia,time and running the program, the profile library will print out a performance summary that looks something like the following:

```
Accelerator Kernel Timing data
c5.c
  test
    32: region entered 1 time
       time(us): total=1411909 init=1408006 region=3903
       kernels=44 data=3859
       w/o init: total=3903 max=3903 min=3903 avg=3903
    35: kernel launched 1 times
       grid: [7x7]  block: [16x16]
       time(us): total=28 max=28 min=28 avg=28
    39: kernel launched 1 times
       grid: [7x7]  block: [16x16]
       time(us): total=16 max=16 min=16 avg=16
```

The information here tells me that the program only executed one accelerator region, at line 32 in routine test in file c5.c. It entered that region only once and spent 1.412 seconds in the region. Of that, 1.40 seconds were spent initializing the device, and 4ms spent executing the kernels in the region. Most of that time was spent moving data, only 44 microseconds spent actually executing kernel code. The region had two kernels, at lines 35 and 39, each executed once.

**Initialization Time**
To optimize the performance of this code, it's clear we want to first address the initialization cost, then try to reduce the time spent moving data. The initialization time here is the time spent in the NVIDIA runtime library and device drivers to initialize memory structures and connect to the device. Normally, this cost only needs to be paid once per program execution. In this case, the initialization is over a second; we've seen it range from 40 milliseconds up to 8 seconds on one cluster installation. For large jobs, this overhead isn't really important, but we'd like to reduce it if it's easy. We're not quite sure why it's so high in some cases and low in others, but we've found that if you have two jobs running in parallel, the initialization for the second job is much less expensive. Apparently most of the time is spent initializing static structures in the device driver, which are kept active as long as there's a process connected to the driver. PGI release 9.0-3 includes a utility program pgcudainit; if you run this program in background mode, it will hold open a CUDA connection to the device driver, significantly reducing initialization time for subsequent programs. For our program, run with pgcudainit activated in background mode, the timing is:

```

 Accelerator Kernel Timing data
c5.c
 test
 32: region entered 1 time
       time(us): total=91628 init=88817 region=2811
       kernels=44 data=2767
 w/o init: total=2811 max=2811 min=2811 avg=2811
 35: kernel launched 1 times
       grid: [7x7] block: [16x16]
       time(us): total=28 max=28 min=28 avg=28
 39: kernel launched 1 times
       grid: [7x7] block: [16x16]
       time(us): total=16 max=16 min=16 avg=16
```

The initialization time was reduced from 1.4 second to less than 1/10 second.

**Host / Accelerator Data Movement**

Now we address the data communication between the host and accelerator. In our example above, data transfer took about 60X as much time as the computation. We should first look at the data transfer \texttt{-Minfo} messages from the compiler; for this program we see:

```
32, Generating copyout(b[1:n-2][1:m-2])
Generating copyin(a[0:n-1][0:m-1])
Generating copyout(a[1:n-2][1:m-2])
```

This example program is the Jacobi iteration example from above. We note two things here. First, the array \texttt{b} is only used as a temporary; its values are not needed back on the host after the loop. As mentioned earlier, the compiler will, by default, always bring modified values back to the host. You can override this by adding an appropriate clause to the accelerator region directive:

```
#pragma acc region local(b[1:n-2][1:m-2])
```

This tells the compiler to allocate the space for the array \texttt{b}, but not to generate any data copies for it. When we build the modified program, the compiler message for the region is:

```
32, Generating local(b[1:n-2][1:m-2])
Generating copyin(a[0:n-1][0:m-1])
Generating copyout(a[1:n-2][1:m-2])
```

When we run this version, the time spent for data traffic in our example program drops from about 2.7ms to 1.7ms.

The second thing we notice is the data traffic for the \texttt{a} array includes a noncontiguous region. The \texttt{copyin} generated is for the whole matrix, but the \texttt{copyout}, from the GPU back to the host, only moves the modified elements, which are the interior of the array. This minimizes the data traffic, but moving noncontiguous regions is more costly than moving one large contiguous section. We can tune this by adding another clause to the region directive:
This tells the compiler to move the whole `a` array both over to the GPU and back again; it moves more data, but the moves are more efficient. The messages from the compiler are now:

```
32, Generating local(b[1:n-2][1:m-2])
Generating copy(a[0:n-1][0:m-1])
```

and the time spent moving data drops from 1.7ms to .5ms. The total savings is 2.7ms to .5ms, about 5X improvement.

The improvement from copying whole arrays depends on the array being allocated in a contiguous block. For C programs, the data layout for multidimensional arrays is under user control. To take advantage of contiguous data transfers, you have to allocate your array as a contiguous memory block. To be specific, suppose you allocate each column of your two-dimensional array separately, as shown here:

```c
b = (float**) malloc( sizeof(float*) * n );
for( i = 0; i < n; ++i )
  b[i] = (float*) malloc( sizeof(float) * m );
```

Because `malloc` adds some control information to the memory blocks it manages, and because it aligns the blocks, two calls to `malloc` never return contiguous memory blocks. This means that your program will never be able to benefit from fast, contiguous DMA transfers between the host and the GPU. Instead, you can allocate the array as follows:

```c
b = (float**) malloc( sizeof(float*) * n );
b[0] = (float*) malloc( sizeof(float) * m * n );
for( i = 1; i < n; ++i ) b[i] = b[i-1] + m;
```

Then the columns are adjacent, and the whole array is a single block of data. Fortran programs don't have this problem, because multidimensional arrays are an intrinsic part of the language.

So three rules for tuning Host–Accelerator data traffic are: First, build with the `-ta=nvidia,time` flag to see whether your program has a data traffic bottleneck. Second, look to see if any arrays are candidates to be local to the accelerator; those arrays would essentially be temporary arrays on the GPU, and are not needed back on the host. Use the `local` clause on the region directive for those arrays. Third, look to see if the compiler is sending noncontiguous subarrays in one direction or the other, and whether it’s feasible to just transfer the whole array in one big chunk. If so, add the appropriate `copy`, `copyin`, or `copyout` clause to the region directive.

**Kernel Performance**

Then we start to look at the performance of the kernel itself. The simple profile information we get back from using the `-ta=nvidia,time` gives no details, so we have to look more deeply. We'll start by looking for compiler messages that give us clues as to the performance, particularly in regards to memory bandwidth utilization on the GPU itself.

An NVIDIA GPU has two kinds of memory. First is the large (up to 4GB, in today’s cards) device memory, which holds all data allocated and initialized by the host. When a GPU kernel is running, data is fetched from the memory in what used to be called superwords; that is, 16 words (64 bytes) are fetched at a time. The threads on the GPU are executed in groups of 32 called warps; data is fetched for half a warp at a time (16 threads). For NVIDIA GPUs with compute capability 1.3, if all the data needed by a half–warp is contained in a single superword, then only that one superword is fetched. For stride–1 array accesses, the 16 threads will access 16 words that will take up one superword (if they are aligned) or two adjacent superwords (otherwise); NVIDIA calls this memory coalescing, when memory accesses from multiple threads are satisfied by one superword fetch. Non–stride–1 accesses will take more superword accesses; in the worst case, strides greater than 16, or indexed memory accesses, it can take up to 16 superword accesses. Since only one word of the 16 actually gets used, this wastes 15/16 (93%) of the available memory bandwidth. So it’s quite important that the kernel be optimized for stride–1 accesses.

Cards with compute capability of 1.2 or lower have more stringent requirements. In such cards, memory coalescing only works when the data access is stride–1 and aligned on a 16–word boundary; that is, the 16
threads in a half-warp must access the 16 memory words in a single superword in the order that the words appear in memory. So it's important to optimize for memory strides, and if possible, for memory alignment. The compiler will do this automatically, as much as it can. When it can't, it gives a message:

32, Non-stride-1 accesses for array 'a'

to flag non-stride-1 accesses; those are cases where you might want to inspect the program to see if you can reorganize the program or change the data layout to remove the non-stride-1 accesses.

The compiler also detects where it can use the software managed data cache to reduce the number of device memory accesses. You'll see a message that looks something like:

34, Cached references to size [16x16] block of 'a'

in those cases.

The current release does not optimize for data alignment. You can sometimes improve the overall performance of your program by padding the leading dimension of your array to a multiple of 16 elements, and using a data directive to send over that whole dimension, even though not all those elements are accessed.

**Kernel Schedule**

Our final step is to tune the kernel schedule. The kernel schedule tells how the parallel loops are mapped onto the hardware parallelism, and it's important to understand. For instance, if there is only a single parallel loop, the compiler will split (strip-mine) the loop into chunks or strips of some length. It will schedule each strip to execute in SIMD or vector mode on a single multiprocessor. It will then schedule the different strips to execute in parallel mode across the multiprocessors. In detail, the compiler will use threadidx.x to index the threads in a single vector strip, and blockidx.x to index the different strips; it will combine these to create the original loop index. In such a case, you'll see a compiler message like:

```
Accelerator kernel generated
33, #pragma for parallel, vector(256)
```

which tells you that the compiler generated vector strips of size 256, and the strips execute in parallel across the multiprocessors. It presents this information using the same directive syntax you would use if you wanted to force this schedule.

If you have two nested parallel loops, the compiler will inspect several loop schedules before choosing one; as mentioned, it tries to optimize memory accesses and total parallelism. Among the options it might choose are:

- Run the inner loop in vector mode and the outer loop in parallel mode:

```
Accelerator kernel generated
33, #pragma for parallel
34, #pragma for vector(256)
```

The inner loop is actually strip-mined in this case as well, in case the trip count is greater than 256, with the strip loop executing sequentially on the GPU.

- Strip-mine the inner loop as before, running the outer loop in parallel mode:

```
Accelerator kernel generated
33, #pragma for parallel
34, #pragma for parallel, vector(256)
```

- Strip-mine both loops, running the strips of each loop in vector mode; note that the product of the vector strip sizes must not exceed 256:

```
Accelerator kernel generated
```
Strip-mine both loops with different strip sizes:

- #pragma for parallel, vector(16)
- #pragma for parallel, vector(16)

- Strip-mine the outer loop for parallel/vector execution, running the inner loop in parallel mode, perhaps to optimize for memory strides:

  Accelerator kernel generated
  - 33, #pragma for parallel, vector(2)
  - 34, #pragma for parallel, vector(64)
  - 33, #pragma for parallel, vector(128)
  - 34, #pragma for parallel

So why might you want to modify the schedule chosen by the compiler? You may know some things that the compiler does not. For instance, the compiler will generally try to generate long vector operations, to take advantage of as much parallelism as possible. Suppose it chooses to run a loop with vector strips of length 256, and you know that usually the trip count of that loop is less than 100; then the compiler is generating useless parallelism. In that case, it might be better to run that loop with a shorter vector length, and perhaps even enable vector parallelism along a second dimension, if available.

There are also performance nonlinearities with today's generation of GPUs. Small changes in the program can make measurable changes in performance. As mentioned above, the current state of the art for scheduling is to search through all the possible loop schedules and test the performance. Until we can train the compiler how to better predict the performance of a kernel, a savvy user will often be able to find a better schedule by trial and error.

**Using cudaprof**

NVIDIA provides a graphical cudaprof performance analysis tool. It runs your program several times and collects time information during the run. When you first fire it up, you'll want to start a new session (File:New), and give it a session name and a directory in which to store the cudaprof project file (suffix .cpj). You then give it the name of your executable file and working directory, any command line arguments, and a maximum time to allow it to execute. You can also select which counters to collect. When you start the program, the default is to run it five times and aggregate the data. The initial screen shows a trace of the calls to NVIDIA runtime routines and GPU kernels.
More interesting are the Summary Table, Width Plot, Height Plot and Summary Plot panes, accessible from the toolbar or under the View menu. The Summary table tells you how many times each kernel and each runtime routine was called, and how much time was spent aggregate in that routine; for the GPU kernels, it also can give a great deal of additional detailed information, such as branches, divergent branches, instruction throughput and more. Here, we see that the program, our Jacobi iteration kernel, is spending most of its time copying data between the host and GPU.
The Width Plot and Height Plot show the program trace in graphical form. The Width Plot shows time horizontally, here showing a couple of kernel calls and many calls to copy data back to the host, because the data is not contiguous.

The Height Plot shows invocation time horizontally and time in that invocation vertically.
Finally, the Summary plot shows what fraction of time was spent in each routine or kernel. We see the effect of adding the copy clauses to optimize the data movement, as described above, with the Summary Plot:
The Height plot shows that there are only three calls to copy data to the device, and one to copy data back, because all the data is now contiguous.

**Summary**

This installment looked at a four step process to tune performance of loops for NVIDIA GPUs: use an appropriate algorithm, tune the host/GPU data movement, optimize for strides and alignment, and understand and perhaps experiment with the kernel schedule. We hope you find the feedback from the compiler useful in your tuning process. We're continuing to work on both the compiler analysis and the feedback, as well as additional features to come in future releases. Our next installment will look in detail at our experiences porting a large application, where we have to address a number of interesting issues.