GPUs have a very high compute capacity, and the latest designs have made them even 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 Toolkit in 2006, including a compiler with extensions to C, that GPU computing became useful without heroic effort. Yet, while CUDA is a big step forward in 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.

In November 2011, PGI, along with Cray, NVIDIA and CAPS Enterprise, introduced the OpenACC API, a directive-based method for host+accelerator programming, allowing us to treat the GPU as an accelerator. The OpenACC specification shares many features with the PGI Accelerator programming model, which PGI introduced in 2009. This article describes OpenACC features for those of you who have little or no experience with GPU programming, and no experience with the PGI Accelerator programming model. If you have used the PGI Accelerator directives, and specifically want information about the differences between the PGI Accelerator directives and OpenACC directives, please refer to the article in the March 2012 issue of the PGIInsider newsletter.

The OpenACC API uses directives and compiler analysis to compile natural C for the GPU; this often allows you to maintain a single source version, because ignoring the directives will compile the same program for the x64 CPU.

Full support for the OpenACC 1.0 specification is included with the PGI 12.6 release of its compilers. Some OpenACC features are available in earlier releases beginning with version 12.3.

**GPU Architecture**

Let’s start by looking at accelerators, GPUs, and the current NVIDIA Fermi GPU in particular, because it was the first target for the PGI 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. Besides GPUs, other accelerators being considered today are the forthcoming Intel MIC, the AMD APUs, and FPGAs. Here we focus on the NVIDIA family of GPUs; a picture of the relevant architectural features is shown below.

The key features are the processors, the memory, and the interconnect. The NVIDIA GPUs have (currently) up to 16 multiprocessors; each multiprocessor has two SIMD units, each unit with 16 parallel thread processors. The thread processors run synchronously, meaning all thread processors in a SIMD unit 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 6GB 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 overwhelm 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 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 2007, 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, the OpenACC consortium has defined the OpenACC API, implemented as a set of directives and API runtime routines. Using OpenACC, you can set up properly, you should see output similar to this:

```
pgaccelinfo
```

You need the right hardware and software to start using the PGI Accelerator compilers. First, you need a 64-bit x86 system with a Linux, MacOS or Windows distribution supported both by PGI and NVIDIA. See the PGI release support page and the Download CUDA page on the NVIDIA website for currently supported distributions. Your system needs a CUDA-enabled NVIDIA graphics or Tesla card; see the CUDA-Enabled Products page on the NVIDIA website for a list of appropriate cards. You need to install the PGI compilers, which include bundled versions of the necessary CUDA toolkit components. From NVIDIA, you'll want a recent CUDA driver, available from the Download CUDA page on the NVIDIA site.

Let's assume you've installed the Linux PGI compilers under the default location /opt/PGI. The PGI installation has two additional directory levels. The first corresponds to the target (linux86 for 32-bit linux86, linux64 for 64-bit). The second level has a directory for the PGI version (12.2 or 12.3 for example) and another PGI release directory containing common components (2012). If the compilers are installed, you're ready to test your accelerator connection. Try running the PGI-supplied tool pgaccelinfo. If you have everything set up properly, you should see output similar to this:

```
CUDA Driver Version: 4010
NVRM version: NVIDIA UNIX x86_64 Kernel Module 285.05.33 Thu Jan 19 14:07:02 PST 2012
Device Number: 0
Device Name: Quadro 6000
Device Revision Number: 2.0
Global Memory Size: 6441992192
Number of Multithreaders: 14
Number of Cores: 448
Concurrent Copy and Execution: Yes
Total Constant Memory: 65536
Total Shared Memory per Block: 49152
Registers per Block: 32768
 Warp Size: 32
Maximum Threads per Block: 1024
Maximum Block Dimensions: 1024, 1024, 64
Maximum Grid Dimensions: 65535 x 65535 x 65535
```

The first line tells you the CUDA driver version information. It tells you that there is a single device, number zero; it's an NVIDIA Quadro 6000, it has compute capability 2.0, 6GB memory and 14 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 tool 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 don't have the right hardware or drivers installed. Presuming everything is working, 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. These example programs are available for download from the PGI website 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 <assert.h>

int main( int argc, char* argv[] )
{
  int n; /* size of the vector */
  float *a; /* the vector */
  float *restrict r; /* the results */
  float *restrict e; /* expected results */
  int i;
  if ( argc > 1 )
    n = atoi( argv[1] );
  else
    n = 10000;
  if ( n <= 0 )
    return 0;

  a = (float*)malloc(n*sizeof(float));
  e = (float*)malloc(n*sizeof(float));
  r = (float*)malloc(n*sizeof(float));
  /* initialize */
  for ( i = 0; i < n; i++ )
    a[i] = (float)(i+1);
  #pragma acc kernels loop
  for ( i = 0; i < n; i++ )
    r[i] = a[i]*2.0f;
  #pragma acc kernels loop
  for ( i = 0; i < n; i++ )
    e[i] = a[i]*2.0f;
  assert( r[i] == e[i] );
  printf( "%d iterations completed\n", n );
  return 0;
}
```

Note the restrict keyword in the declarations of the pointer in the loop; 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[1]*2.0 is computed in double precision, as
((float)(double)a[1] * 2.0). To avoid this, use explicit
float constants, or use the compiler option -Minfo, which
instructs the compiler to treat floating point constants as type
float by default.

We prefixed the loop we want sent to the GPU by a ker-
nels loop directive. This tells the compiler to find the paral-
lelism in the loop, move the data over to the GPU, launch the
operation on the GPU, then bring the results back. For this
program, it’s as simple as that.

Build this with the command:
% pgcc -o acc_c1.exe acc_c1.c -acc -Minfo

Note the -acc and -Minfo flags. The -acc enables the
OpenACC directives in the compiler; by default, the PGI
compilers will target the accelerator regions for an NVIDIA
GPU. We’ll show other options in later examples. The
-Minfo flag enables informational messages from the compi-
er. 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 licensed correctly you should
see the following informational messages from the com-

compiler couldn’t safely determine that
array elements of ar
is assigned, but never read inside
the loop; the values from the CPU memory don’t need to be
to the GPU before kernel execution, but the modified
values need to be copied back. This is a copy-out from the
device memory. The second message
Generating copyout(a[:n])
tells you that the compiler determined that the array a is
used only as input to the loop, so that elements of array
a need to be copied over from the CPU memory to the
GPU device memory; this is a copy-in to the device
memory. Because these elements aren’t modified, they don’t
need to be copied back. 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 exec-
uted in parallel. We added the restrict keyword to the
declarations of the pointer 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
launched on the GPU device memory. You can set the envi-
ronment variable ACC_NOTIFY=1 to 1.

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

Then re-run the program; it will print out a line each time a
GPU kernel is launched. In this case, you’ll see something like:
launch kernel file=acc_c1.c function=main
line=25 device=0 grid=31 block=25
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 environment variable
set for all your programs, but it’s instructive and useful dur-
ing 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,
that has more computational intensity on each itera-
tion. In C, the program is:

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <accel.h>
#include <accelmath.h>

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

    a = (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_rvidia );*/
    gettimeofday( &t1, NULL );

    #pragma acc kernels loop
    for( i = 0; i < n; ++i )
        a[i] = (float)(i+1) * 2.0f;
    gettimeofday( &t2, NULL );
    gcpu = (t2.tv_usec - t1.tv_usec)*1000000 +
          (t2.tv_usec - t1.tv_usec);

    for( i = 0; i < n; ++i )
        a[i] = (float)(i+1) * 2.0f;
    gettimeofday( &t3, NULL );
    cpu = (t3.tv_usec - t2.tv_usec)*1000000 +
          (t3.tv_usec - t2.tv_usec);

    printf(”%d iterations completed\n”, n);
    assert( fabsf(r[i] - e[i]) < 0.000001f );
    printf(”%d iterations completed\n”, n);
    printf(”%d iterations completed\n”, n);
    return 0;
}

The program 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:
• There is a call to acc_init() that is commented out;
you’ll uncomment that shortly.
• There are calls to gettimeofday() to measure wall clock time on the GPU and host loops.
• The program doesn't compare for equality; it compares against a tolerance. We'll discuss that as well.

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 -Winfo by -Winfo=accel on the compile line.

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

```
100000 iterations completed 1359584 microseconds on GPU 40042 microseconds on host
```

So what's this? Over a second on the GPU? Only 1.8 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:

```
other loop that iterates three times, I'll see something more time on the GPU is very long for the first iteration, the first time the of connecting to the GPU on Linux. On Linux, if you run
```

The timing data tells us that the accelerator region at line 32 was entered once and took a total of 1,337 milliseconds. Of that, 1,356 was spent in initialization. The actual execution time was 0.9 milliseconds. Of that time, 530 microseconds was spent moving data back and forth (copying the a and x arrays to and from the GPU), and only 54 microseconds was spent executing the kernel.

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 1360159 microseconds on GPU 1882 microseconds on host
```

The GPU time still includes the overhead of moving data between the GPU and host. Your times may differ, and only 54 microseconds was spent executing the kernel.

The fact is that the GPU doesn't always compute to exactly the same precision as the host. In particular, some transcendental 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 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; 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
void smooth(float* restrict a, float* restrict b, float w1, float w2, float w3, int n, int m, int iter)
{
    int i, j, iter;
    float* tmp;
    for (iter = 1; iter < niter + iter; ++iter)
        #pragma acc kernels loop copyin(b[0:n*m])
    copy(a[0:n*m]) independent
    for (i = 1; i < n-1; ++i) 
        for (j = 1; j < m-1; ++j) 
        {
            a[i*n+j] = w0 * b[i*n+j] + w1 *( b[(i-1)*m+j] + b[(i+1)*m+j] ) + w2 *( b[(i-1)*m+j-1] + b[(i+1)*m+j+1] ) + b[(i)*m+j+1];
            tmp = a;  a = b;  b = tmp;
        }
}
```

Note the use of the restrict keyword on the pointers. 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. We added some clauses to the kernels directive. We added two data clauses; the first is a copyin clause to tell the compiler to copy the array to the GPU.

```
#pragma acc kernels loop copyin(b[(0:0:n*m)])
```

The second is a copy clause to tell the compiler to copy the array starting at \( s[0] \) and continuing for \( n*m \) elements in to the GPU. The second is a copy clause to tell the compiler to copy the array starting at \( t[0] \) and continuing for \( n*m \) elements in to the GPU, and back out of the GPU at the end of the loop. We also added the independent clause to tell the compiler that the iterations of the i loop are data-independent as well, and can be run in parallel.

We'll use the restrict keyword to tell the compiler to copy the array starting at \( s[0] \) and continuing for \( n*m \) elements in to the GPU. The second is a copy clause to tell the compiler to copy the array starting at \( t[0] \) and continuing for \( n*m \) elements in to the GPU, and back out of the GPU at the end of the loop. We also added the independent clause to tell the compiler that the iterations of the i loop are data-independent as well, and can be run in parallel.
We can build this program as usual, and we'll see the information messages:

```
smooth:
31: Generating copyin(b[:m*n])
Generating compute capability 1.0 binary
Generating compute capability 2.0 binary
32: Loop is parallelizable
33: loop is parallelizable
Accelerator kernel generated
32: #pragma acc loop gang, vector(16)
/* blockIdx.x threadIdx.x */
33: #pragma acc loop gang, vector(16)
/* blockIdx.x threadIdx.x */
```

The compiler produces messages about the data copied into and out of the GPU. The next two lines tell us that the compiler generated two GPU binaries, one suitable for Tesla devices (NVIDIA compute capability 1.0-1.3), and a second for Fermi devices (NVIDIA compute capability 2.0). After the line specifying that an accelerator kernel was generated, the compiler gives us the execution schedule, the mapping of the loop iterations to the device parallelism.

```
Generating compute capability 2.0 binary
Generating compute capability 1.0 binary
Generating copy(a[:m*n])
31: Generating copyin(b[:m*n])
```

As before, we see that 90% of the time for the GPU is spent moving data to and from the GPU memory. What if, instead of moving the data back and forth for each iteration, we could move the data once and leave it there? OpenACC gives us a way to do just that. Let's modify the program so in the routine that calls smooth, we surround that call with a data construct:

```
#pragma acc data copyin(b[0:n*m], a[0:n*m])
{
  smooth(a, b, w0, w1, w2, n, n, iters);
}
```

This tells the compiler to copy both arrays to the GPU before the call, and bring the results back to the host memory after the call. Inside the function, replace the copyin and copy data clauses by a present clause:

```
#pragma acc kernel loop present (b[0:n*m], a[0:n*m]) independent
```

This tells the compiler that the data is already present on the GPU, so rather than copying the data it should just use the copy that is already present. In the output from the modified program below, we see no data movement in the smooth function, just the 16.5 milliseconds for kernel execution.

```
Accelerator Kernel Timing data
acc_c3a.c
smooth
31: region entered 50 times
time(us): total=172415 init=18
region=172397
 kernel=16849 data=143582
w/o init: total=172397 max=3942
min=3425 avg=3447
33: kernel launched 50 times
grid: [63x63] block: [16x16]
time(us): total=16849 max=355
min=331 avg=336
acc_init.c
acc_init
30: region entered 1 time
time(us): init=137202
```

But suppose we want a program that will run on the GPU when it's available, or on the host when it's not. The PGI compilers provide that functionality using the PGI Unified Binary™ technology. To enable it, build with the `target accelerator option: -ta=nvidia,host`. This generates two versions of the smooth function, one that runs on the host and one on the GPU. 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:
27: PGI Unified Binary version for -tp=nehalem-64 -ta=host
33, 2 loop-carried redundant expressions removed with 2 operations
and 4 arrays
```

Summary
This article introduced OpenACC features in the PGI Accelerator C compiler for NVIDIA GPUs, and presented three simple programs. We looked at issues you may encounter with float vs. double and unrestricted pointers. We enabled OpenACC directives with the `-acc` flag, used the simple accelerator profile library with the PGI ACC_TIME environment variable, and used `-ta=nvidia,host` to generate a unified host-GPU binary. We also briefly introduced the data construct and the present clause. We encourage you to try these examples to get started programming GPUs with OpenACC.
by Michael Wolfe, PGI Compiler Engineer

General purpose parallel programming on GPUs is a relatively recent phenomenon. GPUs were originally hardware blocks optimized for a small set of graphics operations. As demand arose for more flexibility, GPUs became increasingly more programmable. Early approaches to computing on GPUs cast computations into a graphics framework, allocating buffers (arrays) and writing shaders (kernel functions). Several research projects looked at designing languages to simplify this task; in late 2006, NVIDIA introduced CUDA and tools to make data parallel computing on a GPU more straightforward. Not surprisingly, the data parallel features of CUDA map pretty much the same, where some cores are disabled for conditional operations.

The code is actually executed in groups of 32 threads, what NVIDIA calls a warp. A Fermi multiprocessor takes two cycles to execute one instruction for an entire warp on each group of 16 cores, for integer and single precision operations. For double precision instructions, a Fermi multiprocessor combines the two groups to look like a single 16-core double precision multiprocessor (instead of two groups of 16 cores); this means the double precision throughput is half of the single precision throughput.

There is also a small software-managed data cache attached to each multiprocessor, shared among the cores; NVIDIA calls this the shared memory. This is a low-latency, high-bandwidth, indexable memory which runs close to register speeds. On Fermi, the shared memory is actually 64KB, and can be configured as a 48KB software-managed data cache with a 16KB hardware data cache, or the other way around (16KB SW, 48KB HW cache).

When the threads in a warp issue a device memory operation, that instruction will take a very long time, perhaps hundreds of clock cycles, due to the long memory latency. Mainstream architectures include a two-level or three-level cache memory hierarchy to reduce the average memory latency, and Fermi does include some hardware caches, but mostly GPUs are designed for stream or throughput computing, where cache memories are ineffective. Instead, these GPUs tolerate memory latency by using a high degree of multithreading. A Fermi supports up to 48 active warps on each multiprocessor. When one warp stalls on a memory operation, the multiprocessor control unit selects another ready warp and switches to that one. In this way, the cores can be productive as long as there is enough parallelism to keep them busy.

Programming

NVIDIA GPUs are programmed as a sequence of kernels. Typically, each kernel completes execution before the next kernel begins, with an implicit barrier synchronization between kernels. Fermi has some support for multiple, independent kernels to execute simultaneously, but most kernels are large enough to fill the entire machine. As mentioned, the multiprocessors execute in parallel, asynchronously. However, GPUs do not support a fully coherent memory model that would allow the multiprocessors to synchronize with each other. Classical parallel programming techniques can’t be used here. Threads can’t spawn more threads on the current GPUs; threads on one multiprocessor can’t send results to threads on another multiprocessor; there’s no facility for a critical section among all the threads across the whole system. Trying to use a Tthreads or OpenMP programming model will lead to pain, frustration, and failure.

CUDA offers a data parallel programming model that is supported on NVIDIA GPUs. In this model, the host program launches a sequence of kernels. A kernel is organized as a hierarchy of threads.

Threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid. Kernels can use these indices to compute array subscripts, for instance.

Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can
synchronize and share data with threads in the same block; a warp will always be a subset of threads from a single block. Threads in different blocks may be assigned to different multiprocessors concurrently, to the same multiprocessor concurrently (using multithreading), or may be assigned to the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically.

There is a hard upper limit on the size of a thread block, 1,024 threads or 32 warps for Fermi. Thread blocks are always created in warp-sized units, so there is little point in trying to create a thread block of a size that is not a multiple of 32 threads; all thread blocks in the whole grid will have the same size and shape. A Fermi multiprocessor can have 1,536 threads simultaneously active, or 48 warps. These can come from 2 thread blocks of 24 warps, or 3 thread blocks of 16 warps, 4 thread blocks of 12 warps, and so on up to 8 blocks of 6 warps; there is another hard upper limit of 8 thread blocks simultaneously active on a single multiprocessor.

Performance tuning on NVIDIA GPUs requires optimizing all these architectural features:

• Finding and exposing enough parallelism to populate all the multiprocessors.

• Finding and exposing enough additional parallelism to allow multithreading to keep the cores busy.

• Optimizing device memory accesses for contiguous data, essentially optimizing for stride-1 memory accesses.

• Utilizing the software data cache to store intermediate results or to reorganize data that would otherwise require non-stride-1 device memory accesses.

This is the challenge for the CUDA programmer, and for the PGI Accelerator compilers. If you are a CUDA Fortran programmer, we hope this gives you a basic understanding that you can use to tune your kernels and launch configurations for efficient execution.

If you are an OpenACC programmer, this article should give you some background for understanding the compiler feedback messages and making efficient use of the loop mapping directives. The OpenACC directives are designed to allow you to write concise, efficient and portable x64+GPU programs. However, writing efficient programs still requires you to understand the target architecture, and how your program is mapped onto the target. The OpenACC directives are designed to reduce the cost of writing simple GPU programs, and to make it much less tedious to write large efficient programs. We hope this article is one step towards that understanding.

5x in 5 Hours: Porting a 3D Elastic Wave Simulator to GPUs Using OpenACC

by Mathew Colgrove, PGI Applications Engineer

In September 2011, PGI presented the PGI Accelerator programming model at the Society of Exploration Geophysics (SEG) annual meeting in San Antonio, TX. Scientists in the oil and gas industry often investigate large yet highly parallelizable problems that can adapt well to accelerators.

As an example case, we looked at Seismic CPML (http://www.geodynamics.org/cig/software/seismic_cpm) developed by Dimitri Komatitsch and Roland Martin from University of Pau, France. From their website: “SEISMIC_CPM is a set of ten open-source Fortran 90 programs to solve the two-dimensional or three-dimensional isotropic or anisotropic elastic, viscoelastic or poroelastic wave equations using a finite-difference method with Convolutional or Auxiliary Perfectly Matched Layer (C-PML or ADE-PML) conditions.”

In particular, we decided to accelerate the 3D Isotropic application which is a 3D elastic finite-difference code in velocity and stress formulation with Convolutional-PML (C-PML) absorbing conditions. In addition to being highly compute intensive, the code uses MPI and OpenMP to perform the domain decomposition, giving us an opportunity to showcase the use of multi-GPU programming.

As an example case, we looked at Seismic CPML (http://www.geodynamics.org/cig/software/seismic_cpm) developed by Dimitri Komatitsch and Roland Martin from University of Pau, France. From their website: “SEISMIC_CPM is a set of ten open-source Fortran 90 programs to solve the two-dimensional or three-dimensional isotropic or anisotropic elastic, viscoelastic or poroelastic wave equations using a finite-difference method with Convolutional or Auxiliary Perfectly Matched Layer (C-PML or ADE-PML) conditions.”

In particular, we decided to accelerate the 3D Isotropic application which is a 3D elastic finite-difference code in velocity and stress formulation with Convolutional-PML (C-PML) absorbing conditions. In addition to being highly compute intensive, the code uses MPI and OpenMP to perform the domain decomposition, giving us an opportunity to showcase the use of multi-GPU programming.

This article was first published in the March 2012 issue of the PGInsider newsletter and focused on applying PGI Accelerator directives to SEISMIC_CPM. Recently we updated the code to use OpenACC directives (a ten minute process) and this version of the article reflects those changes.
like the average intensity to be above 4, but 2.5 is sufficient to move forward.

Seismic_CPML uses MPI to decompose the problem space across the Z dimension. This will allow us to utilize more than one GPU, but it also adds extra data movement as the program needs to pass halos (regions of the domain that overlap across processes). We could use OpenMP threads as well, but doing so would add more programming effort and complexity to our example. As a result, we chose to remove the OpenMP code from the GPU version. We may revisit that decision in a future article.

As an aside, with the MPI portion of the code the programmer manually decomposes the domain. With OpenMP, the compiler does that automatically if the program is running in a shared memory environment. Currently, OpenMP compilers aren’t able to automatically decompose a problem across multiple discrete memory spaces as is the case when using multiple devices. As a result, the programmer must manually decompose the problem just like they would using MPI. Because this would basically require us to duplicate our effort, we decided to forgo using OpenMP here.

Below is the initial timing for the original MPI/OpenMP code on our test system. Note that while we have presented these results previously, all timings have been refreshed for this article.

### Version 2.0 Processes

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>GPUs</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Host</td>
<td>2</td>
<td>4</td>
<td>0.933</td>
</tr>
</tbody>
</table>

Problem Size: 1041x128
System Information: 2 Core Intel Core-i7 920 Running at 2.67Ghz with 2 Tesla C2070 GPU
Compiler: PGI 2012 version 12.5

**Step 1: Adding Setup Code**

Because this is an MPI code where each process will use its own GPU, we need to add some utility code to ensure that happens. Current generation Fermi cards don’t support multiple host processes using the same GPU at the same time. While it may work in some cases, this practice is not recommended. The `setDevice` routine first determines which node the process is on (via a call to `hostid`) and then gathers the hostids from all other processes. It then determines how many GPUs are available on the node and assigns the devices to each process.

Note that in order to maintain portability with the CUDA version, this section of code is guarded by the preprocessor macro `_OPENACC`, which is defined when the OpenACC directives are enabled in the PGI Fortran compiler through the use of the `-acc` command-line option.

```
#ifdef _OPENACC
    function setDevice(nprocs,myrank)
        integer :: nprocs, myrank
        integer :: hostid, ierr, numdev,
        integer, dimension(nprocs) :: hostids,
        integer :: localprocs, mydev
        ! get the number of devices on this node
        hostid = gethostid()
        if (hostid .eq. hostids(i)) then
            do i=1,nprocs
                localprocs=0
                if (localprocs .eq. eq, hostids()) then
                    do i=numlocal to numlocal+1
                        localprocs = localprocs + 1
                    enddo
                    if (numdev > localprocs) then
                        print *, 'ERROR: Number of processes is greater than number of GPUs.', myrank
                        checkError('setDevice', ierr)
                        mydev = 0
                    else
                        mydev = mod(localprocs+1, numdev)
                    endif
                    return
                endif
            enddo
        else
            print *, 'ERROR: There are no devices available on this host. & ABORTING.', myrank
            checkError('setDevice', ierr)
            mydev = -1
        endif
    end function setDevice
endif
```

**Step 2: Adding Compute Regions**

Next, we spent a few minutes adding six compute regions around the eight parallel loops. For example, here’s the final reduction loop.

```
!$acc kernels
do k = km, knax
    do j = nipoints_m, 1, -1
        do i = nipoints_m, 1, -1
            compute kinetic energy first, defined as
            (1/2 rho ||v||^2); in principle we should use
            rho_half_k_half_y instead of rho for v
            ! in order to interpolate density at the right
            ! location in the staggered grid cell but in a
            ! homogeneous medium we can safely ignore it
            total_energy_kinetic += &
            total_energy_potential += &
            2.d0*epsilon_xz*sigmaxz(i,j,k) + &
            2.d0*epsilon_yz*sigmayz(i,j,k) + &
            2.d0*epsilon_yy*sigmayy(i,j,k) + &
            2.d0*epsilon_zz*sigmazz(i,j,k) + &
            2.d0*epsilon_xz*sigmaxz(i,j,k) + &
            2.d0*epsilon_yz*sigmayz(i,j,k) + &
            2.d0*epsilon_yy*sigmayy(i,j,k) + &
            2.d0*epsilon_zz*sigmazz(i,j,k) + &
            2.d0*epsilon_xz*sigmaxz(i,j,k)
        enddo
    enddo
enddo
```

The `-acc` command line option to the PGI Accelerator Fortran compiler enables OpenACC directives. Note that OpenACC is meant to model a generic class of devices. While NVIDIA is the current market leader in HPC accelerators and default target for PGI’s OpenACC implementation, the model can, and will in the future, target other devices.
Another compiler option you’ll want to use during development is -Minfo, which causes the compiler to output feedback on optimizations and transformations performed on your code. For accelerator-specific information, use the -Minfo=accel sub-option. Examples of feedback messages produced when compiling Seismic_CPML include:

```
1113, Generating copyin(xyz(11:91,11:631,kmin:kmax))
Generating copyin(sigmayz(11:91,11:631,kmin:kmax))
Generating copyin(sigmaxz(11:91,11:631,kmin:kmax))
Generating copyin(sigmazz(11:91,11:631,kmin:kmax))
Generating copyin(sigmayy(11:91,11:631,kmin:kmax))
Generating copyin(vz(11:91,11:631,kmin:kmax))
```

To compute on a GPU, the first step is to move data from host memory to GPU memory. In the example above, the compiler tells you that it is copying over nine arrays. Note the `copyin` statements. These mean that the compiler will only copy the data to the GPU but not copy it back to the host. This is because line 1113 corresponds to the start of the reduction loop compute region, where these arrays are used but never modified. Other data movement clauses you may see include `copy` where the data is copied to the device at the beginning of the region and copied back at the end of the region, and `copyout` where the data is only copied back to the host.

Notice that the compiler is only copying an interior subsection of the arrays. By default, the compiler is conservative and only copies the data that’s actually required to perform the necessary computations. Unfortunately, because the interior sub-arrays are not contiguous in host memory, the compiler needs to generate multiple data transfers for each array. Overall GPU performance is determined largely by how well we can optimize memory transfers. That means not just how much data gets transferred, but how many transfers occur. Transferring multiple sub-arrays is very costly. For now we will just note it. Later, we'll look at improving performance by overriding the compiler defaults and copying entire arrays in one large contiguous block.

Here the compiler has performed dependence analysis on the loops at lines 1114, 1115, and 1116 (the reduction loop shown earlier). It finds that all three loops are parallelizable so it generates an accelerator kernel. When the compiler encounters loops that cannot be parallelized, it generally reports a reason why so that you can adapt the code accordingly. If inner loops are not parallelizable, a kernel may still be generated for outer loops; in those cases the inner loop(s) will run sequentially on the GPU cores.

The compiler may attempt to work around dependences that prevent parallelization by interlooping loops (i.e. changing the order) where it’s safe to do so. At least one outer or interchanged loop must be parallel for an accelerator kernel to be generated. In some cases you may be able to use the loop directive independent clause to work around potential dependences, or the private clause to eliminate a dependence entirely. In other cases, code may need to be restructured significantly to enable parallelization.

The generated accelerator kernel is just a serial bit of code that gets executed on the GPU by many threads simultaneously. Every thread will be executing the same code but operating on different data. How the threads are organized is called the loop schedule. Below we can see the loop schedule for our reduction loop. The do loops have been replaced with a two-dimensional gang, which in turn is composed of a three-dimensional vector section.

```
1114, Loop is parallelizable
1115, Loop is parallelizable
1116, Loop is parallelizable
```

In CUDA terminology, the `gang` clause corresponds to a grid dimension and the `vector` clause corresponds to a thread block dimension. For new or non-CUDA programmers, we highly recommend reading Michael Wolfe’s preceding article “Understanding the Data Parallel Threading Model for GPUs”.

Don’t feel too overwhelmed by loop schedules. It’s just a way to organize how the GPU threads act on data elements of an array. So here we have a 3-D array that’s being grouped into blocks of 16x4x4 elements where a single thread is working on a specific element. Because the number of gangs is not specified in the loop schedule, it will be determined dynamically when the kernel is launched.

If the `gang` clause had a fixed width, such as `gang(16)`, then each kernel would be written to loop over multiple elements.

With CUDA, programming reductions and managing shared memory can be a fairly difficult task. In the example below, the compiler has automatically generated optimal code using these features. By the example below, the compiler has automatically generated optimal code using these features. By the example below, the compiler has automatically generated optimal code using these features.

```
1120, Sum reduction generated for total_energy_kinetic
1140, Sum reduction generated for total_energy_potential
```

OK, so how are we doing?

Don’t get discouraged if this happens to you. So why the slowdown and how can we fix it?

**Step 3: Adding Data Regions**

You may have guessed that the slowdown is caused by excessive data movement between host memory and GPU memory. In looking at a section of our CUDA profile information, we see that each compute region is spending a lot of time copying data back and forth between the host and device. Because each compute region is executed 2500 times, this makes for a lot of data movement. Adding up all the data transfer times in the profile output for Step 2 shows that of the total 3031 seconds, almost 2800 were spent copying data while only about 100 were spent in compute kernels. The remaining time is spent either in host code or blocked waiting on data transfers (both MPI processes must use the same PCIe bus to transfer data).

```
seismic_cpm1_3d_iso_mpi_acc
1113: region entered 2505 times
  time(us): total=238341920 init=990
  region=238340930
  kernel=7943489
  data=217581655
  w/o init: total=238340930 max=122566
  min=90870 avg=95336
1116: kernel launched 2500 times grid: [16x64x1] block: [16x4x4]
  time(us): total=7900622 max=3263
  min=3046 avg=3160
1140: kernel launched 2500 times grid: [2] block: [256]
  time(us): total=42886 max=27
  min=16 avg=17
```

On a side note, the above profile information was obtained by setting the environment variable `PGI_ACC_TIME=1` and running our executable. This option prints basic profile information such as the kernel execution time, data transfer time, initialization time, the actual launch configuration, and total time spent in a compute region. Note that the total time is measured from the host and includes time spent executing host code within a region.
For later steps, we also will use profile information obtained from the CUDA driver by setting the CUDA_PROFILE environment variable to 1. The profile is output to a log file and lists the times for each data transfer and kernel launch. We then use an included perl script, totalProf.pl, to aggregate the times. A third profiling option, not used here, is the PGI pgcollect utility, which collects both host and GPU profile times and allows you to view these times in the PGPROF performance profiler tool.

To improve performance, we need to find a way to minimize the amount of time transferring data. Enter the data directive. You can use a data region to specify exact points in your program where data should be copied from host memory to GPU memory, and back again. Any compute region enclosed within a data region will use the previously copied data, without the need to copy at the boundaries of the compute region. A data region can span across host code and multiple compute regions, and even across subroutine boundaries.

In looking at the arrays in Seismic_CMPL, there are 18 arrays with constant values. Another 21 are used only previously copied data, without the need to copy at the boundaries of the compute region. A data region can span across host code and multiple compute regions, and even across subroutine boundaries.

Let's start by adding a data region around the outer time loop. The final three arrays do need to be copied back to the host to pass their halos. For those cases, we use the update directive.

```fortran
!$acc data (vz)
... do it = 1,NSTEP
!$acc update host (sigmazz,sigmays,sigmazx)
  sigmazz(k1), left shift
call MPI_SENDRECV(sigmazz(:,1,:), &
  number_of_values, MPI_DOUBLE_PRECISION, &
  receiver_left_shift, message_tag, &
  sigmazz(:,1:Z_LOCAL+1), number_of_values, &
  ...) !$acc end data
```

Data regions can be nested, and in fact we used this feature in the time loop body for the arrays vx, vy, and vz as shown below. While these arrays are copied back and forth at the inner data region boundary, and so are moved more often than the arrays moved in the outer data transfer, they are used across multiple compute regions instead of being copied at each compute region boundary. Note that we do not specify any array dimensions in the copy clause. This instructs the compiler to copy each array in its entirety as a contiguous block, and eliminates the inefficiency we noted earlier when interior sub-arrays were being copied in multiple blocks.

```fortran
!$acc data copy(vx,vy,vz)
... data region spans over 5 compute regions and host code
```

We're making good progress, but we can improve the performance even further.

### Step 4: Optimizing Data Transfers

For our next step we'll work to optimize the data transfers even further by migrating as much of the computation as we can over to the GPU and moving only the absolute minimum amount of data required. The first step is to move the start of the outer data region up so that it occurs earlier in the code, and to put the data initialization loops into compute kernels. This includes the vx, vy, and vz arrays. Using this approach enables us to remove the inner data region used in our previous optimization step.

In the following example code, notice the use of the clausenew directive or a copy clause at a data or compute region boundary.

```fortran
!$acc data

How are we doing on our timings? This step took just about an hour of coding time and reduced our execution time down to 550 seconds, of which only 400 seconds are now spent transferring data. The overall compute kernels time is holding steady at 100 seconds.

<table>
<thead>
<tr>
<th>Version</th>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>GPUs</th>
<th>Execution Time (sec)</th>
<th>Programming Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Host</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>951</td>
<td>-</td>
</tr>
<tr>
<td>ACC Step 2</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>3001</td>
<td>10</td>
</tr>
<tr>
<td>ACC Step 3</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>590</td>
<td>-</td>
</tr>
</tbody>
</table>

Problem Size: 10x4x128
System Information: 4 Core Intel Core-i7 920 Running at 2.67Ghz with 2 Tesla C2070 GPUs
Compiler: PGI 2012 version 12.5

We're making good progress, but we can improve the performance even further.

### Step 4: Optimizing Data Transfers

For our next step we'll work to optimize the data transfers even further by migrating as much of the computation as we can over to the GPU and moving only the absolute minimum amount of data required. The first step is to move the start of the outer data region up so that it occurs earlier in the code, and to put the data initialization loops into compute kernels. This includes the vx, vy, and vz arrays. Using this approach enables us to remove the inner data region used in our previous optimization step.

In the following example code, notice the use of the create clause. This instructs the compiler to allocate space for variables in GPU memory for local use but to perform no data movement on those variables. Essentially they are used as scratch variables in GPU memory.

```fortran
!$acc data

One caveat to using data regions is that you must be aware of which copy (host or device) of the data you are actually using in a given loop or computation. The host and device copies of the data are not automatically kept coherent. That is the responsibility of the programmer when using data regions. For example, any update to the copy of a variable in device memory won’t be reflected in the host copy until you specify that it should be updated using either an update directive or a copy clause at a data or compute region boundary.
Unintentional loss of coherence between the host and device copy of a variable is one of the most common causes of validation errors in OpenACC programs. After making the above change to Seismic_CPML, the code generated incorrect results. After nearly a half hour of debugging, we determined that the section of the time step loop that initializes boundary conditions was omitted from an OpenACC compute region. As a result we were initializing the host copy of the data, rather than the device copy as intended, which resulted in uninitialized variables in device memory.

The next challenge in optimizing the data transfers related to the handling of the halo regions. Seismic_CPML passes halos from six 3-D arrays between MPI processes during the course of the computations. Ideally we would simply copy back the 2-D halo sub-arrays using update directives or copy clauses, but as we saw earlier copying non-contiguous array sections between host and device memory is very inefficient. As a first step, we tried copying the entire arrays from device memory back to the host before passing the halos. This was also very inefficient, given that only a small amount of the data moved between host and device memory was needed in the eventual MPI transfers.

After some experimentation, we settled on an approach whereby we added six new temporary 2-D arrays to hold the halo data. Within a compute region we gathered the 2-D halos from the main 3-D arrays into the new temporary arrays. This was also very inefficient, given that only a small amount of the data moved between host and device memory was needed in the eventual MPI transfers.

In the example code below, note that the source code added to support the halo gathers and transfers is guarded by the preprocessor _OPENACC macro and will only be executed if the code is compiled by an OpenACC-enabled compiler.

```c
#include _OPENACC
!

The above modifications required about two hours of coding time, but as reflected in the table below our total execution time is now down to 124 seconds with only 5 seconds spent copying data! The kernel execution time has increased slightly to 106 seconds due to the added work on the GPU, but overall we achieved a nice speed-up with this change.

The above modifications required about two hours of coding time, but as reflected in the table below our total execution time is now down to 124 seconds with only 5 seconds spent copying data! The kernel execution time has increased slightly to 106 seconds due to the added work on the GPU, but overall we achieved a nice speed-up with this change.

<table>
<thead>
<tr>
<th>Version</th>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>GPUs</th>
<th>Execution Time (sec)</th>
<th>Programming Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Host</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>951</td>
<td>-</td>
</tr>
<tr>
<td>ACC Step 2</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>3031</td>
<td>10</td>
</tr>
<tr>
<td>ACC Step 3</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>550</td>
<td>60</td>
</tr>
<tr>
<td>ACC Step 4</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>124</td>
<td>120</td>
</tr>
</tbody>
</table>

### Step 5: Loop Schedule Tuning

The final step in our tuning process was to tune the OpenACC compute region loop schedules using the 

vector clauses. In many cases, the default kernel schedules chosen by the PGI OpenACC compiler are quite good. Manual tuning efforts often don’t improve timings significantly. However, in some cases the compiler doesn’t do as well. It’s always worthwhile to spend a little time examining whether you can do better by overriding compiler-generated loop schedules using explicit loop scheduling clauses. You can usually tell fairly quickly if the clauses are having an effect.

Unfortunately, there is no well-defined method for finding an optimal kernel schedule (short of trying all possible schedules). The best advice is to start with the compiler’s default schedule and try small adjustments to see if and how they affect execution time. The kernel schedule you choose will affect whether and how shared memory is used, global array accesses, and various types of optimizations. Typically, it is more optimal to perform gang scheduling of loops with large iteration counts. In the case of Seismic_CPML, the outer loop has the smallest iteration count so we tried exchanging one of the gang clauses between the k and j loops. In this case, we saw a slight improvement with the following schedule:

```c
!$acc do vector(4) do k = k2begin,NZ_LOCAL
  kglobal = k + offset_k
!$acc do vector(4) do j=1,NY

which resulted in the following timings:

<table>
<thead>
<tr>
<th>Version</th>
<th>MPI Processes</th>
<th>OpenMP Threads</th>
<th>GPUs</th>
<th>Execution Time (sec)</th>
<th>Programming Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Host</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>951</td>
<td>-</td>
</tr>
<tr>
<td>ACC Step 2</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>3031</td>
<td>10</td>
</tr>
<tr>
<td>ACC Step 3</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>550</td>
<td>60</td>
</tr>
<tr>
<td>ACC Step 4</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>124</td>
<td>120</td>
</tr>
</tbody>
</table>

### Conclusion

In a little over five hours of programming time we achieved about a 7x speed-up over the original MPI/OpenMP version running on our test system. On a much larger cluster running a much larger dataset (see below), speed-up is not quite as good. There is additional overhead in the form of inter-node communication, and the GPUs on the system have more cores and run at a higher clock rate. Nonetheless, the speed-up is still nearly 5x.
The enclosed articles were originally published in the *PGInsider* newsletter. These and many more like them are designed to help you effectively use PGI compilers and tools. The *PGInsider* is free to subscribers.

Register on the PGI website at [http://www.pgroup.com/register](http://www.pgroup.com/register)

PGFORTRAN, PGI Unified Binary and PGI Accelerator are trademarks and PGI, PGI CDK, PGCC, PGC++, PGI Visual Fortran, PVF, Cluster Development Kit, PGPROF, PGDBG and The Portland Group are registered trademarks of The Portland Group, Incorporated, a wholly-owned subsidiary of STMicroelectronics, Inc.

© 2009-2012 The Portland Group, Inc. All rights reserved.