OpenACC 2.0

PGI®

Using the OpenACC Routine Directive

by Michael Wolfe, PGI Compiler Engineer

One of the key features of high-level language programming is modularity, including support for procedures and separate compilation. It's hard to imagine modern programming without functions and libraries. The term compiler was originally used to define the software that compiled separately created external objects into a single binary, what we now call a linker. Yet, until recently, OpenACC programs could only support procedures through inlining, more or less preventing any use of libraries or procedure calls across multiple files. With the latest releases, PGI now supports procedure calls, separate compilation and linking for OpenACC programs targeting NVIDIA GPU accelerators.

This article introduces this very important feature and explains how to use the **`acc routine`** directive to enable it. It also presents hints on how to use the available clauses on the routine directive, explanations for why the clauses are necessary, and some caveats and current limitations.

A Simple Example

Let's start with an example in C, with one file containing a function:

```
#include <math.h>
#pragma acc routine seq
float sqab(float a){
    return sqrtf(fabsf(a));
}
```

The **`acc routine`** pragma directs the compiler to generate accelerator code for the following routine, in addition to the usual host code generation. The **`seq`** clause informs the compiler that the code in the routine, and in any other routines called from within this routine, will run sequentially in one device thread on an accelerator. We'll have more details on this clause later in the next section. This function can be compiled as follows:

```
% pgcc -acc -Minfo -c a1.c
sqab: 4, Generating acc routine seq
         Generating Tesla code
```

This compiles the function for host execution as expected, and also for execution on an accelerator. The informational messages generated by the compiler confirm that the routine was compiled for a Tesla accelerator, the default with the PGI compilers when the **`-acc`** option is specified. Both versions of the function are placed in a standard host object file, and **`sqab()`** can be called either from host code or from an OpenACC compute region.

We can call this function within an accelerator region in another routine in a separate file:

```
#pragma acc routine seq
extern float sqab(float);
...
void test(float* x, int n){
      #pragma acc parallel loop pcopy(x[0:n])
      for( int i = 0; i < n; ++i ) x[i] = sqab(x[i]);
}
```

The function `test()` includes a simple OpenACC parallel loop with a call to `sqab()`; the prototype for `sqab()` appears in the file, and a routine directive is inserted before the prototype to inform the compiler that an accelerator version of the function is available, and to specify the conditions under which it can be called. Alternatively, the routine directive could have been specified anywhere after the prototype by including the function name:

```
#pragma acc routine(sqab) seq
```

This form is useful if the prototype appears in a header file. Compiling `test()` with the same command line flags as above will give the following informational messages:

```
test: 6, Generating present_or_copy(x[:n])
         Accelerator kernel generated
      7, #pragma acc loop gang, vector(256)
         /*blockIdx.x threadIdx.x */
      6, Generating Tesla code
```

Because the program has both host code and accelerator device code that must be linked, the PGI compiler will invoke the device linker followed by the host linker, and embed the device binary into the host binary. You can encounter link-time errors for device code, just as for host code, if your program includes a device-side call to a missing routine. In this case, if we compile the routine `test()` as above without the routine directive for `sqab()`, we see a link-time error:

```
nvlink error : Undefined reference to 'sqab'in'a2.o'
```

In this scenario, `sqab` had been compiled for host execution but not for device execution, so an undefined reference error issues from the device-side linker. We can write the same example in Fortran. Here we put the function in a module:

```
module b1
contains
   real function sqab(a)
      !$acc routine seq
      real :: a
      sqab = sqrt(abs(a))
   end function
end module
```

In Fortran the routine directive is placed inside the function or subroutine, usually in the declarations section. A caller can then simply use the module:

```
subroutine test( x, n )
   use b1
   real, dimension(*) :: x
   integer :: n
   integer :: i
   !$acc parallel loop pcopy(x(1:n))
   do i = 1, n
      x(i) = sqab(x(i))
   enddo
end subroutine
```

The Routine Directive

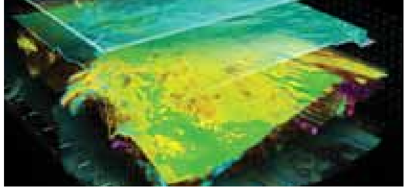
As shown above, the **acc routine** directive has two uses. First, it directs the compiler to compile the current routine for device execution as well as host execution. As we describe below, it also gives very specific information about how the routine may be called on an accelerator device. This is done in C or C++ by placing an **acc routine** directive just before the function; the directive may specify the function name in parentheses after the **routine** keyword, or not. In a Fortran subroutine or function, this is done by placing an **acc routine** directive (with or without the procedure name) in the declarations section of the procedure.

The second use for the directive is to convey to the compiler that a device version of a called routine exists, possibly in another file, and to further convey how it will be compiled. In a C or C++ program, if the function appears in the same file above the call site, the routine directive just above that routine will suffice; no additional directive is needed.

```
#pragma acc routine seq
float sqab( float a ){
    return sqrtf( fabsf( a ) ); }
...
void test( float* x, int n ){
    #pragma acc parallel loop pcopy(x[0:n])
    for( int i = 0; i < n; ++i ) x[i] = sqab( x[i] );
}
```

If the function appears later in the file, or is in another file, you can place an **acc routine** directive just before a prototype declaration for the function or an **acc routine(*name*)** directive anywhere in the file above the call site.

```
    // routine pragma immediately before the prototype:
#pragma acc routine seq
extern float sqab(float);
   // or, after the prototype but before the call site:
#pragma acc routine(sqab) seq
...
```



```
void test( float* x, int n ){
    #pragma acc parallel loop pcopy(x[0:n])
    for( int i = 0; i < n; ++i ) x[i] = sqab( x[i] );
}
...
  // repeat routine pragma before the function itself:
#pragma acc routine seq
float sqab( float a ){
    return sqrtf( fabsf( a ) );
}
```

In a Fortran program, if the called routine is in a module any subprogram that uses the module will have the necessary information, as in our example above (another reason to use Fortran modules). Otherwise, the caller can include an interface block for the routine, with an **acc routine** directive that matches the one in the actual routine:

```
subroutine test( x, n )
   real, dimension(*) :: x
   integer :: n
   interface
      real function sqab(a)
      !$acc routine seq
      real :: a
      end function
   end interface
   ...
end subroutine
```

Alternatively, the caller can include an external declaration for the routine and an **acc routine(*name*)** directive:

```
subroutine test( x, n )
   real, dimension(*) :: x
   integer :: n
   real, external :: sqab
   !$acc routine(sqab) sqab
   ...
end subroutine
```

For C++, many functions appear as methods in class declarations. Many of these methods are in header files, often in files that can't be modified. Moreover, these classes have templates and can be instantiated many times. To address these complexities, the PGI C++ compiler has two additional features.

First, a routine directive for a templated class member function that appears in the class definition will apply to any instantiation of that function. Second, any function in the same file (or an included file) that is called from within an OpenACC compute region is compiled with an implicit **`acc routine seq`** directive. This includes class member functions, as well as functions called from within other functions that have an explicit or implicit routine directive. This functionality is specifically to address the prevalent definition of classes in header files, so is limited to C++ at this time.

Routine Parallelism Clauses

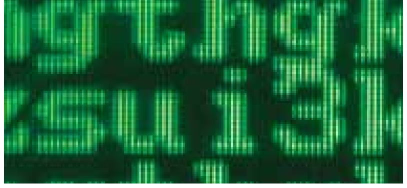
The **`acc routine`** directive should include a clause indicating the kind of parallelism used in the program unit to which it applies. This will be one of gang, worker, vector or seq. An **`acc routine gang`** directive informs the compiler that the routine contains a loop with gang parallelism, or calls another routine that has a gang parallel loop. Such a routine may not itself be called from within a gang parallel loop, because OpenACC gang parallel loops cannot be nested. Because gang parallel loops are executed by sharing the iterations of the loop across the gangs, all gangs must make the call to such a routine. For you OpenMP programmers, this is essentially the same as the OpenMP restriction that a procedure with an orphan **`for`** or **`do`** directive on a loop must be called by all OpenMP threads.

An **`acc routine worker`** directive tells the compiler that the routine contains worker parallelism, but no gang parallelism. Such a routine may be called from within a gang parallel loop, but not within a worker or vector parallel loop. Similarly, an **`acc routine vector`** directive tells the compiler that the routine contains vector parallelism, but no gang or worker parallelism. It may be called from within a gang parallel or worker parallel loop, but not from within a vector parallel loop.

Finally, an **`acc routine seq`** directive tells the compiler that the routine contains no parallelism; it is a sequential routine that will execute only on the vector lane of the one worker of the one gang that called the routine.

A savvy programmer will ask why OpenACC needs these clauses when OpenMP does not. It's true that OpenMP 3.1 does not require any directives when compiling routines. This is partly because OpenMP 3.1 programs are only compiled for a single instruction set, so the parallel code runs on the same processors as the sequential code, (i.e. on the host). In addition, OpenMP 3.1 only has a single level of parallelism, corresponding to OpenACC gang parallelism. If there is only a single level of parallelism, then a routine may contain a parallel loop or not. If not, then it's a sequential routine. If it does, then it's the programmer's responsibility to ensure all OpenMP threads actually make the call so the parallel loop is properly work-shared across all the threads.

However, the new OpenMP 4 **`simd`** capability adds the same complexity, and requires programmers to declare when a procedure must be compiled for SIMD execution. The OpenMP 4 **`simd`** functionality corresponds roughly to OpenACC vector mode, and so the two specifications now have similar requirements when it comes to addressing function calls and multiple levels of parallelism.



Routine with Vector Parallelism

Here we show an example of a Fortran module with a routine declared with vector parallelism.

```
module j1
contains
subroutine saxpy(n,a,x,y)
   !$acc routine vector
   integer,value :: n
   real :: a, x(*), y(*)
   integer :: i
   !$acc loop
   do i = 1, n
      y(i) = a*x(i) + y(i)
   enddo
end subroutine

subroutine test( x, y, a, n, gpu )
   real :: x(:,:), y(:,:), a(:)
   integer :: n
   integer :: i
   logical :: gpu
   !$acc parallel loop pcopy(y) pcopyin(x,a) if(gpu)
   do i = 1, n
      call saxpy( n, a(i), x(1,i), y(1,i) )
   enddo
end subroutine
end module
```

The `saxpy` routine is declared with `acc routine vector`. This means that vector parallelism will be exploited within the routine, not by the caller. When the compiler sees the orphaned `acc loop`, it knows to use only vector parallelism for that loop, because of the routine directive. The loop directive is called orphaned because it is not contained in an OpenACC parallel or kernels construct; it will be in a compute region generated in the caller.

The parallel loop directive in the caller does not have an explicit `vector_length` clause. However, because the compiler knows that this construct contains a call to the `saxpy` subroutine which was compiled with the `routine vector` attribute, the compiler knows that vector parallelism should be generated. For NVIDIA devices, the PGI compiler will add an implicit `vector_length(32)` clause. In this example, the conditional `if(gpu)` clause is used to determine whether to run the compute region on the accelerator device (in this case a GPU) or on the host. This can be very useful for testing, to compare results generated on an accelerator with results generated by the same code executed on the host.

Routine with Gang Parallelism

This final example is a variation on the last one, modifying subroutine test to run on the device

```
module jj1
contains
subroutine saxpy(n,a,x,y)
   !$acc routine vector
   integer,value :: n
   real :: a, x(*), y(*)
   integer :: i
   !$acc loop vector
   do i = 1, n
      y(i) = a*x(i) + y(i)
  enddo
end subroutine

subroutine test( x, y, a, n )
   !$acc routine gang
   real :: x(:,:), y(:,:), a(:)
   integer, value :: n
   integer :: i
   !$acc loop gang
   do i = 1, n
      call saxpy( n, a(i), x(1,i), y(1,i) )
   enddo
end subroutine
end module
```

The vector **`saxpy`** routine hasn't changed. The subroutine **`test`** now has a routine directive. The **`acc parallel loop`** becomes another orphaned **`acc loop`** directive in this example. In this case, the caller must specify the number of gangs, because the compiler won't know the trip count of the gang loop. The caller must give an explicit vector length as well, because the compiler doesn't see the call to the vector routine:

```
!$acc parallel pcopy(y) pcopyin(x,a) num_gangs(n) &
!$acc vector_length(32)
call test( x, y, a, n )
!$acc end parallel
```

Notice we changed the scalar argument **`n`** to have the **`value`** attribute, so it will be passed efficiently by value. In C and C++, all arguments are passed by value; arrays are supported by passing the value of the pointer to the array. In Fortran, arguments are usually passed by reference, that is, by passing the address of the argument. For scalars, this is inefficient; Fortran now supports the **`value`** attribute for scalar dummy arguments. To use this, the caller must have an explicit interface for the procedure, usually by putting the caller and the procedure in a module, or putting the procedure in a module and using that module in the caller.

Disabling Relocatable Code

By default, the PGI compilers generate relocatable device code for Tesla targets. The device linker is invoked before the host linker as described earlier, and a device binary is generated at link time allowing for separate compilation of source files. You can disable separate compilation if all your procedure calls appear in the same source file by using the **`nordc`** suboption (no relocatable device code) on the compiler command line:

```
    -ta=tesla:nordc
```

In **`nordc`** mode, the compiler generates a device binary for each source file. Calls to procedures that don't appear in that source file will result in undefined references at compile time. If you specify **`-ta=tesla:nordc`** for the link step, the device linker will not be invoked. In that case, all the OpenACC files must have been compiled with **`nordc`**. In some cases, there is a performance advantage to disabling relocatable code because the device compiler back-end will do more function inlining and optimization without relocatable code.

There is one other reason to disable relocatable code for Tesla targets. With relocation disabled, the CUDA binary includes code for all the specified compute capabilities, and also includes PTX (portable assembler). If you build with **`-ta=tesla:cc20,nordc`**, the CUDA binary will include Fermi code as well as PTX. When you run your program on a Kepler or newer GPU, the PTX will be dynamically recompiled for the newer machines. With the current PGI compilers, the portable PTX code is lost when generating relocatable code and linking. This is being addressed for a future release.

The **`acc routine`** directive and procedure calls can be used with Radeon targets as well, but AMD does not currently support separate compilation for Radeon accelerators. There is currently no capability to link Radeon objects, though we expect that feature to be available in the future. For Radeon, the caller and callee routines must currently be in the same source file. If you specify both Tesla and Radeon targets, **`-ta=radeon,tesla`**, the compiler will assume the **`nordc`** suboption for the Tesla target as well.

Compute Capabilities

The current default compute capability suboption for Tesla targets is **`-ta=tesla:cc2+`**, which means it will generate code for Fermi (cc20) and Kepler (cc30) targets. If you specify the compute capability for the compile step, you should use the same compute capabilities at the link step. In particular, if you compile with **`-ta=tesla:cc30`**, you won't be able to link without specifying the same option. By default, the compiler will try to link with **`cc20`** and **`cc30`**, and it will find no **`cc20`** code in the object file.

If you get a link-time error message:

```
    nvlink fatal : could not find compatible device codein c.o
```

then likely you need to specify the proper compute capability at the link step.

Current Limitations

There are some current limitations in support for procedures. We've already discussed the lack of separate compilation for Radeon targets. For NVIDIA accelerator devices, a procedure compiled with **`acc routine vector`** will usually only work if called from a parallel construct with **`vector_length(32)`**. The OpenACC vector loops must synchronize across all the vector lanes at the end of the loop. OpenACC workers and vector lanes map to dimensions of a single thread block, and the CUDA execution model doesn't allow synchronizing an arbitrary subset of the thread block, such as those CUDA threads comprising the vector lanes of a single worker. This limitation currently restricts **`routine vector`** usage to a vector width that maps to a single CUDA warp.

Automatic local arrays in Fortran and VLAs (variable length arrays) in C are not supported in OpenACC routines with the PGI 14.x releases. Fixed size arrays are of course supported. Also, passing Fortran assumed shape dummy arguments is not supported in OpenACC routines with PGI 14.x releases.

Closing Comments

With support for separate compilation, OpenACC can now support true high level programming. Object files containing both host and device code can be placed in libraries and the appropriate code will be linked as expected. Part 2 of this article will discuss more advanced issues, including support for global variables and the **`acc declare`** directive, using the **`acc routine`** directive to interface with CUDA C and CUDA Fortran functions, and includes examples of using **`acc routine`** in C++ classes.

OpenACC Routine Directive Part 2

by Michael Wolfe, PGI Compiler Engineer

The previous article introduced the OpenACC routine directive and its use to enable true procedure calls and separate compilation in OpenACC programs. This article will discuss a few more advanced issues: support for global variables and the `acc declare` directive, interfacing to CUDA C and CUDA Fortran device functions using `acc routine` declarations, and using `acc routine` in C++ class member functions.

Global Variables in C and C++

As with external routines, external variables can be referenced in OpenACC accelerator regions and device code. As with routines, a directive is required to inform the compiler that a global variable needs to be allocated in device memory as well as host memory. Let's assume we have a global variable `coef` containing a global coefficient that we want to use in device code.

```
float coef = 3.14159265f;
#pragma acc declare copyin(coef)
#pragma acc routine seq
float adjust(float a){
    return a*a/coef;
}
```

The `acc declare` directs the compiler to generate a global copy of `coef` in the device code. The `copyin` clause will cause the value of `coef` on the host to be copied to the device when the program attaches to the device. This typically occurs when the program executes its first OpenACC data or compute construct, or when `acc_init` is called.

The declare directive must appear in any file that refers to the variable in device code; it must also appear in the file that actually declares the variable without the `extern` keyword. Suppose we have this modified example:

```
extern float coef;
#pragma acc declare copyin(coef)
#pragma acc routine seq
float adjust(float a){
    return a*a/coef;
}
```

Just as the extern `coef` must be declared in another file for the host code, that file must have a declare directive to put `coef` on the device. This example used a `copyin` clause; the `acc declare` directive can also use the `create` clause, which will allocate the variable but not initialize it from the host.

Note that the `copyin` clause will cause the value to be copied from the host to the device when the device is attached. The host program can set or modify the value before that time, and the modified value will be copied to the device. The global variable can be used directly in OpenACC compute regions:

```
extern float coef;
#pragma acc declare copyin(coef)
...
#pragma acc parallel
{
   float y = sin(coef);
   #pragma acc loop
   for( i = 0; i < n; ++i ) x[i] *= y;
}
```

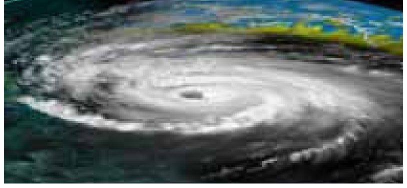
Global variables also appear in the OpenACC runtime present table. This means that you can use the **`update`** directive to copy updated values of the variable between the host and device memories. Also, global arrays can be passed to procedures that use the array in a **`present`** data clause:

```
void vec_double( float* x, int n ){
   #pragma acc parallel loop present(x[0:n])
   for( int i = 0; i < n; ++i )
      x[i] += x[i];
}
...
float gx[1024];
#pragma acc declare create(gx)
...
vec_double( &gx[1], 1022 );
#pragma acc update host( gx )
```

Currently, C file static variables, function static variables, and C++ class static members can appear in **`acc declare create`** directives, and will be allocated on the device, but they will not appear in the present table and cannot appear in **`update`** directives.

There is an additional benefit when the global variable is itself a pointer. Typically, we don't want just the pointer on the device, we want the data that the pointer points to as well. With the PGI implementation, if you have a global pointer in an **`acc declare`** directive, then list an array based on that pointer in a data clause, the compiler allocates and copies the array to the device, as directed in the data clause, and fills in the global pointer on the device with the device array address.

In the example that follows, the pointer **`p`** is a global variable in both host and device memories. At the enter data directive, the array **`p[0:n]`** is copied to device memory, and the global pointer **`p`** on the device is updated to point to this array. In the routine **`addem`**, the global pointer **`p`** will have the correct value.



```
float* p;
#pragma acc declare create(p)
...
void addem( float* x, int n ){
    #pragma acc parallel loop present(x[0:n])
    for( i = 0; i < n; ++i ) x[i] += p[i];
}
...
float *xx;
p = (float*)malloc( n*sizeof(float) );
#pragma acc enter data copyin( p[0:n] )
#pragma acc data copy( xx[0:n] )
{
 ...
 addem( xx, n );
 ...
}
```

Module Variables in Fortran

In Fortran, global variables are in modules or in common blocks. Unfortunately, common blocks are not currently supported in accelerator device code because of device linker limitations. However, module variables are supported. A directive in the module declaration informs the compiler that a variable needs to be allocated in device memory as well as host memory.

```
module globals
    real :: coef = 3.14159265
    !$acc declare copyin(coef)
contains
real function adjust( a )
    real :: a
    adjust = a*a / coef
end function
end module
```

The `acc declare` directs the compiler to generate a copy of `coef` in device memory. The `copyin` clause will cause the value of `coef` on the host to be copied to the device when the program attaches to the device. Because this appears in a module, `coef` can be used in host or device code in any subprogram in the module, or in any subprogram that uses the module.

```
use globals
...
!$acc parallel loop
do i = 1, n
    x(i) = x(i) * coef
enddo
```

As in C, these variables appear in the present table, and can be used in `update` directives.

There is one other use for `acc declare` in a module. An allocatable array that appears in an `acc declare create` clause will be allocated on the device as well as the host when it appears in an allocate statement. If the allocatable array is declared in a module, the global device copy of the allocatable array pointer gets updated with the device address at the allocate statement. Such an array will appear in the present table, and values can be moved between host and device memories with `update` directives.

In the following example, a module contains an allocatable array that appears in a `declare create` directive. The allocate statement in the `allocx` routine allocates `x` in both host and device memory. The call to `muly` in the parallel construct will occur on the device, using the device copy of `x.` Note the scalar arguments to `muly` are declared with the `value` attribute; a more efficient parameter passing mechanism than the Fortran default of passing by reference.

```
module globals
    real, dimension(:), allocatable :: x
    !$acc declare create(x)
contains
subroutine muly( y, a, n )
    !$acc routine vector
    real, dimension(*) :: y
    real, value :: a
    integer, value :: n
    integer :: i
    !$acc loop vector
    do i = 1, n
        x(i) = x(i) + y(i)
    enddo
end subroutine
end module
subroutine allocx( n )
    use globals
    integer :: n
    allocate( x(n) ) ! allocates host and device copies
end subroutine
subroutine doem( y, a, n )
    use globals
    real, dimension(:) :: y
    real :: a
    integer :: n
    integer :: i
    !$acc data copyin(y)
    !$acc parallel num_gangs(1) vector_length(128)
    call muly( y, a, n )
    !$acc end parallel
    !$acc end data
end subroutine
```

Calling CUDA Device Routines from C

With separate compilation on Tesla accelerators, you might want to write a CUDA device routine that can be called from your OpenACC compute construct or from within a procedure compiled with **acc routine**. The PGI OpenACC compiler maps parallel loop iterations onto the CUDA **blockIdx** and **threadIdx** indices.

The simplest case is a scalar CUDA C device routine, one that doesn't refer to **threadIdx** or **blockidx** indices, such as:

```
__device__ __host__ float radians( float f ){
    return f*3.14159265;
}
```

When compiling this device routine with **nvcc**, you must specify **-rdc=true** and the compute capability (or capabilities) that you are compiling for:

```
 % nvcc -c -rdc=true -gencode arch=compute_35,code=sm_35
```

The next step is to add **acc routine seq** after the prototype for this device routine in your OpenACC source code. The **seq** clause tells the compiler that each device thread will call this routine independently of any other thread.

```
extern float radians( float );
#pragma acc routine(radians) seq
```

The **radians** routine may then be called within an OpenACC compute region. Because the original routine had both the **__device__** and **__host__** attributes, it can be called on either the CUDA device or on the host. If the routine didn't have the **__host__** attribute, the OpenACC program would have to be compiled with **-ta=tesla** explicitly, leaving off the **-ta=host** option. If such a routine were called within another procedure with **acc routine**, the routine directive should include the **nohost** clause, because there is no host version of **radians** to call.

If your CUDA device routine expects all the device threads to call it, in particular if your CUDA device routine includes calls to **__syncthreads**, the **acc routine** directive for the prototype should use the **worker** clause, instead of **seq**. Such a routine will typically use **blockIdx** and **threadIdx** to determine on which indices to work. The **worker** clause directs the compiler to ensure all threads in the corresponding thread block actually call the procedure simultaneously. The PGI OpenACC compilers typically use **threadIdx.x** to compute the vector loop index and **threadIdx.y** to compute the worker loop index, so the CUDA device routine should use that as well.

Calling CUDA Device Routines from Fortran

In CUDA Fortran, a device routine is defined with **attributes(device)** on the subroutine or function statement. If a scalar device routine appears in a module, and the OpenACC routine uses that module, the device routine may be called directly in an OpenACC compute construct. Device procedures can be called within OpenACC compute constructs, just as device data can be used or assigned; as mentioned earlier, these constructs must be compiled with

-ta=tesla explicitly, and in particular without the **-ta=host** option, because there is no host version of the CUDA device data or device procedure.

The simplest way to use CUDA Fortran device routines is to place them in a module, then call them from the same module or use that module in the caller. In the example below, note again the use of the **value** attribute on the scalar argument **f**.

```
module m1
contains
attributes(device) real function radians( f )
    real, value :: f
    radians = f*3.14159265
end function
end module
subroutine sub( x )
    use m1
    real, dimension(:) :: x
    integer :: i
    !$acc parallel loop present(x)
    do i = 1, ubound(x,1)
       x(i) = radians(x(i))
    enddo
end subroutine
```

Calling CUDA C routines from Fortran OpenACC compute constructs requires an interface block:

```
subroutine sub( x )
    real, dimension(:) :: x
    interface
        real function radians( f ) bind(c)
        !$acc routine seq
        real, value :: f
        end function
    end interface
    integer :: i
    !$acc parallel loop present(x)
    do i = 1, ubound(x,1)
       x(i) = radians(x(i))
    enddo
end subroutine
```

The **routine seq** directive indicates that this routine will have been compiled for the device. The **bind(c)** effects Fortran code generation using C bindings, meaning the compiler doesn't decorate the name in the generated code as it would for a Fortran function name. Rather, it uses symbol names and calling conventions expected for a C function.

C++ Class Member Functions

In a C++ program, many functions, especially class member functions, appear as source code in header files included in the program. The PGI C++ compiler will take note of functions called in compute regions and implicitly add the pragma **`acc routine seq`** if there is no explicit **`routine`** directive. With optimization, many of these functions will get inlined anyway, but this allows the program to compile without having to modify header files, many of which are read-only system header files.

If you have your own class definition and want to add an explicit **`acc routine`** directive, do so just above the function definition in the class. This will allow the compiler to generate a device version of the member function, so it can be called on an accelerator device. If the class is templated, the compiler will generate a version for each instantiation of that class. Note that virtual functions are not supported on the device, nor is exception handling. The example below creates a routine **`incrby`** that can be called on the device using vector parallelism.

```
template<class T>class myv{
    T* x;
    int n;
public:
    T& operator[](int i){ return *(this->x+i); }
    int size(){ return n; }
    ...
    #pragma acc routine vector
    void incrby( myv& b ){
        int nn = n;
        #pragma acc loop
        for( int i = 0; i < nn; ++i ) x[i] += b[i];
    }
    ...
}

void test( int n ){
    myv<float> a(n);
    myv<float> b(n);
    ...
    #pragma acc parallel num_gangs(1) vector_length(100)
    {
        a.incrby( b );
    }
}
```

The instantiation of **`myv`** for the variables **`a`** and **`b`** will create a version of the **`incrby`** function for the device. Of equal interest is how to create the copy of the **`myv`** variables **`a`** and **`b`** on the device, along with their data. The OpenACC committee is working on several ways to handle this, but for now you can write class member routines similar to a constructor and destructor, that create or delete the device data, and update the device data or the host data as in the following example.

```
template<class T>class myv{
    T* x;
    int n;
public:
 void createdev(){
     #pragma acc enter data create( this[0:1], x[0:n] )
     #pragma acc update device(n)
 }
 void deletedev(){
     #pragma acc exit data delete( x[0:n], this[0:1] )
 }
 void updatedev(){
     #pragma acc update device(x[0:n])
 }
 void updatehost(){
     #pragma acc update host(x[0:n])
 }
}
```

The create routine `createdev` creates the class itself (through the `this` pointer) and the data, using the OpenACC 2.0 dynamic data `enter data` directive. The OpenACC runtime will allocate memory for the class, then memory for the data vector `x`, then fill in or attach the pointer from the class to the data vector. It also explicitly fills in the length field. The delete routine deletes the data vector and the class, in reverse order. The routines to update the data simply update the data vector in one direction or the other.

Closing Comments

The routine directive supports a `nohost` clause that directs the compiler to generate only a device version of a function or procedure. In particular, no host version will be generated. This obviously only works if there are no calls to the routine from host code, so this clause should be used with care. Also supported is a `bind` clause, which allows you write a completely different implementation of a routine for use on an accelerator device versus the host. Because this clause is being re-defined for the next revision of the OpenACC specification, I'll leave discussion of it until the definition is settled.

The OpenACC routine directive makes it reasonably natural to write parallel programs that use modern modular programming structures, including separate compilation and libraries. I hope this two-part introduction gets you started to more productive use of OpenACC for developing performance portable heterogeneous HPC applications.

PGI C++ and OpenACC

by Michael Wolfe, PGI Compiler Engineer

C++ programmers often lament that most examples in OpenACC tutorials and presentations use Fortran or C. Because C++ is (almost) a superset of C, one could argue that these are in fact all legal C++ examples, but that's like saying that Ford Model T automobiles came in your choice of color. C++ is designed for higher levels of data abstraction and code reuse than are possible with either C or Fortran, particularly with templated classes. This article discusses how to use such datatypes with OpenACC and the PGI compilers. It also covers features that work today, features that will work in early 2015, and features being designed for future revisions of OpenACC.

Introductory Example

Let me start with a simple C program using OpenACC directives, to set the stage and to compare and contrast with the C++ analog. Note, this is not an introduction or tutorial about OpenACC directives and programming. If you need an introduction to GPU programming or OpenACC directives, check out the PGI website at www.pgroup.com/openacc.

This example isn't very sophisticated, nor is it particularly useful, but it will demonstrate some important differences between C and C++. The **axpy** function below is essentially the BLAS **daxpy** routine (A times vector X plus vector Y). The caller places the data on the device, calls **axpy**, updates one of the vectors in some way, calls the function again, then stops.

```
void axpy( double* y, double* x, double a, int n ){
   #pragma acc parallel loop present(x[0:n],y[0:n])
   for( int i = 0; i < n; ++i )
      y[i] += a*x[i];
}
void test( double* a, double* b, int n ){
   #pragma acc data copy( a[0:n] ) copyin( b[0:n] )
   {
      axpy( a, b, 2.0, n );
      modify( b, n );
      #pragma acc update device(b[0:n])
      axpy( a, b, 1.0, n );
   }
}
```

Three OpenACC directives are used to control data movement and effect parallel execution of the loop on the accelerator device. The **#pragma acc data** directive in the caller directs the compiler to allocate space for the **a** and **b** vectors and to copy the data for both from the host to the device memory. That construct starts at the directive and ends at the matching closing brace. At the end, the updated **a** vector will be copied back to the host memory, but because the **b** vector was in a **copyin** clause, the host data will not be updated. Reducing the data moved between host and device like this can be an important performance optimization, though in this little program it won't have much of an effect.

There are two calls to `axpy`, and between them the vector `b` is updated on the host using the routine `modify()` and the updated values are transferred to the device with the `#pragma acc update device` directive. Inside routine `axpy`, the `#pragma acc parallel loop` directs the compiler to generate code to execute the iterations of that loop in parallel on the accelerator using the data for arguments `x` and `y` that are already present there.

We can compile this example using the PGI C++ compiler (or the C compiler, but we'll use C++ throughout) enabling OpenACC with the `-acc` flag as follows:

```
% pgc++ -acc -Minfo=accel -O2 -Kieee -o s1 s1.cpp
```

The `-acc` option enables OpenACC directives processing and accelerator code generation; the default accelerator target for the PGI compilers is an NVIDIA GPU. The `-Minfo=accel` generates informational messages from the compiler, as we shall see shortly. The `-O2` option sets the optimization levels, and the `-Kieee` option disables any optimizations that don't strictly follow the IEEE rules for computation or normalization. In this case, it prevents use of the fused multiply-add instructions on either the host (if available) or the accelerator.

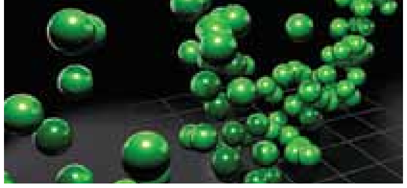
The `-Minfo=accel` informational messages provide details on how the compiler processed the OpenACC directives. In this case, we see the messages:

```
axpy(double *, double *, double, int):
  16, Generating present(x[:n])
      Generating present(y[:n])
      Accelerator kernel generated
      18, #pragma acc loop gang, vector(256)
          /* blockIdx.x threadIdx.x */
  16, Generating Tesla code
test(double *, double *, int):
  25, Generating copy(a[:n])
      Generating copyin(b[:n])
  31, Generating update device(b[:n])
```

The messages indicate the compiler has scheduled the parallel loop at line 18 across both gang parallelism (thread blocks for NVIDIA GPUs) and vector parallelism (threads within a thread block). They also describe the data movement for the data construct at line 25 and the update directive at line 31. Running this program on a machine with an NVIDIA GPU should produce the same result as running the program without directives on the host.

Using C++ Classes and Templates

But, as I admitted earlier, that was really a C program. How about changing this to take advantage of the C++ encapsulation features. What if instead of a naked C pointer to an array of floats, I build a vector class? You might well ask why I don't just use the standard template library vector class, and that's a valid and very good question. The short answer is that support for `std::vector` class demands more features in the OpenACC language and compilers than are available today; I'll have more detailed information about that later in this article.



For now, I'll build my own vector class, which will allow us to highlight some of the new PGI compiler features for C++ class members. I'll call it **`myvector`** with two data members: a private data pointer and a size field.

```
template<typename vtype>
   class myvector{
   vtype* mydata;
   size_t mysize;
public:
```

To this I need to add a constructor and destructor and a couple of member functions, like the operator **`[]`** and a **`size`** access function:

```
inline vtype & operator[]( int i ) const {return mydata[i];}
inline size_t size(){ return mysize; }
myvector( int size_ ){
    mysize = size_;
    mydata = new vtype[mysize];
    #pragma acc enter data copyin(this)
    #pragma acc enter data create(mydata[0:mysize])
}
~myvector(){
#pragma acc exit data delete(mydata[0:mysize],this)
delete [] mydata;
}
```

For this example, the constructor allocates data on the host using **`new`** and allocates on the device as well using OpenACC 2.0 **`enter data`** directives. The first directive allocates memory for the struct **`myvector`** itself and initializes it with the host values. The value for the **`mydata`** pointer won't be of much use, but we want the **`mysize`** field correctly initialized on the device. The second directive allocates device memory for the data vector; this directive will also fill in the **`mydata`** pointer in the device copy of the **`myvector`** struct. The destructor deletes the device memory for the data vector as well as the **`myvector`** struct. We can add two more member functions to update the device and host data vectors:

```
void updatedev(){
    #pragma acc update device(mydata[0:mysize])
}
void updatehost(){
    #pragma acc update host(mydata[0:mysize])
}
};
```

With this class defined, we can create an **`axpy`** function that takes two templated vectors and a scalar:

```
template<typename vtype> void
axpy( myvector<vtype>& y, myvector<vtype>& x, vtype a ){
    size_t n = y.size();
    #pragma acc parallel loop present(x,y)
    for( int i = 0; i < n; ++i )
        y[i] += a*x[i];
}
```

The caller now changes as well:

```
template<typename vtype> void
test( myvector<vtype>& a, myvector<vtype>& b ){
    a.updatedev();
    b.updatedev();
    axpy<vtype>( a, b, 2.0 );
    modify( b );
    b.updatedev();
    axpy<vtype>( a, b, 1.0 );
    a.updatehost();
}
```

Because the data for the vectors `a` and `b` has already been allocated by their constructors, we only need to update the device data using our `updatedev` and `updatehost` member functions. If we compile this with the same pgc++ command line as above with our C example, we'll see informational messages that look as follows:

```
void axpy<double>(myvector<t1> &, myvector<t1> &, T1):
  42, Generating present(x[:])
      Generating present(y[:])
      Accelerator kernel generated
      44, #pragma acc loop gang, vector(256)
          /* blockIdx.x threadIdx.x */
  42, Generating Tesla code
myvector<double>::operator [](int) const:
  18, Generating implicit acc routine seq
      Generating Tesla code
myvector<double>::myvector(int):
  25, Generating enter data copyin(this[:1])
      Generating enter data create(mydata[:mysize])
myvector<double>::~myvector():
  28, Generating exit data delete(this[:1])
      Generating exit data delete(mydata[:mysize])
myvector<double>::updatedev():
  32, Generating update device(mydata[:mysize])
myvector<double>::updatehost():
  35, Generating update host(mydata[:mysize])
```

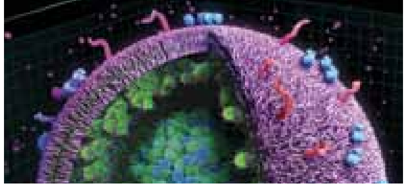
It's interesting to note that the messages aren't sorted by line number. The compiler builds the program from the top-down, from caller to callee. The **axpy** and **test** routines are external routines, so must be compiled, but the class member functions are only compiled once the data type has been instantiated, and in fact may be compiled several times, once for each type instantiation.

Also interesting is the message **Generating implicit acc routine seq** for the **operator []**. This operator is invoked both on the host as well as on the device. It will usually be inlined, but because it's called in device code the compiler generates a device version of the operator in case inlining is disabled. The PGI C++ compiler does this for any class member function, and in fact for any function at all that is called in a compute region.

Running this program is more or less the same as running the C version. One of the features of the PGI implementation of OpenACC is the ability to trace the OpenACC activity and to summarize time spent copying data and executing on the accelerator. Setting the environment variable **PGI_ACC_NOTIFY** to **3** before executing a program will cause the OpenACC runtime libraries to print a line before each data movement or accelerator kernel launch. Setting it to **1** will only print kernel launches, and to **2** will only print upload and download lines. Setting it to **3** and running this program, we see the following:

```
upload CUDA data  file=s2.cpp function=_ZN8myvectorIdEC1Ei
line=25 device=0 variable=_T33828688 bytes=16
upload CUDA data  file=s2.cpp function=_ZN8myvectorIdEC1Ei
line=25 device=0 variable=.pointer. bytes=8
upload CUDA data  file=s2.cpp function=_ZN8myvectorIdEC1Ei
line=25 device=0 variable=_T33828688 bytes=16
upload CUDA data  file=s2.cpp function=_ZN8myvectorIdEC1Ei
line=25 device=0 variable=.pointer. bytes=8
upload CUDA data  file=s2.cpp function=_
ZN8myvectorIdE9updatedevEv line=32 device=0 bytes=8000
upload CUDA data  file=s2.cpp function=_
ZN8myvectorIdE9updatedevEv line=32 device=0 bytes=8000
launch CUDA kernel  file=s2.cpp function=_
Z4axpyIdEvR8myvectorIT_ES3_S1_ line=42 device=0 num_gangs=4
num_workers=1 vector_length=256 grid=4 block=256
...
```

The first upload line corresponds to the **enter data copyin(this)** for the constructor for **a**. The next line is the update of the 8-byte pointer to the data vector at the **enter data create(mydata)** directive on the next line. These two lines are repeated for the **b** constructor. The next two upload lines are for the **update device** in the **updatedev()** function. The function names here are mangled according to the GCC mangling rules. If we run this through the PGI utility **pggdecode**, we will see the un-mangled names:



```
function=myvector<double>::myvector(int) line=25
function=myvector<double>::updatedev() line=32
function=void axpy<double>(myvector<t1> &,
                           myvector<t1> &, T1) line=42
```

One question you might ask is why should the **axpy** function not use **y.size()** as the loop limit, instead of saving it into the variable **n**? This is a limitation in the PGI 14.9 compiler, which won't run that loop in parallel if the loop limit is a reference to a class member.

More Data Control

One problem with the previous approach is that it allocates all vectors on the device. Because accelerator memory is typically more limited than host memory, we may not want to put all data in both host and device memory. We can modify this class to remove device data management from the constructor and destructor and create member functions to handle that behavior. In the example below, the constructor and destructor handle only host data. The **devcopyin**, **devcopyout** and **devdelete** member functions manage device data. The **devcopyin** function allocates and initializes the device copy of the data from the host; **devcopyout** copies the device data back to the host and deletes it; **devdelete** deletes the device data, assuming the host copy is already up-to-date or doesn't need to be updated.

```
myvector( int size_ ){
    mysize = size_;
    mydata = new vtype[mysize];
}
~myvector(){
    delete [] mydata;
}
void devcopyin(){
    #pragma acc enter data copyin(this,mydata[0:mysize])
}
void devcopyout(){
    #pragma acc exit data copyout(mydata[0:mysize])
    #pragma acc exit data delete(this)
}
void devdelete(){
    #pragma acc exit data delete(mydata[0:mysize],this)
}
```

Finally, instead of a separate **axpy** function, we can make it a member function as well:

```
void axpy( myvector<vtype>& x, vtype a ){
    #pragma acc parallel loop present(this,x)
    for( int i = 0; i < mysize; ++i )
        mydata[i] += a*x[i];
}
```

With this modified **myvector** class, we need to modify the calling routine:

```
    template<typename vtype> void
    test( myvector<vtype>& a, myvector<vtype>& b ){
        a.devcopyin();
        b.devcopyin();
        a.axpy( b, 2.0 );
        modify( b );
        b.updatedev();
        a.axpy( b, 1.0 );
        a.devcopyout();
        b.devdelete();
}
```

Here it's assumed the vectors **a** and **b** are not already present on the device. The calls to **devcopyin** will allocate and copy those vectors to the device. At the end, the call to **a.devcopyout** will bring the updated values of **a** back to the host and free the device memory, and **b.devdelete** will simply free the device copy of **b**.

We can build with **pgc++** as before, and we'll get similar informational messages. You may ask why the **present(this)** is on the parallel loop directive in the member function **axpy**. In this loop, **mydata** is implicitly referenced through the implicit **this** argument pointer. It would be equivalent to put **mydata[0:mysize]** in the **present** clause.

Future Development

Why can't the program simply use the existing **std::vector** class? The whole discussion may contain more details than you want to know, but I'll present as much information as I can without getting too technical. One problem is the **vector** class typically appears in a system header file that can't be modified. This means that solutions like we present here, adding device management directives and/or functions, is a non-starter. One potential solution under discussion within the OpenACC committee is a mechanism to apply directives to a class as if they had been specified within the scope of that class, even though the directives appear in another header file.

A second problem is that a typical implementation of **std::vector** has three data members: a pointer to the start of the vector data, a pointer to the end of the vector data, and a pointer to the end of the allocated space for the vector. There is no member that corresponds to the size of the vector. When copying the vector class to a device, we want to allocate space for the vector data, then fill in all three pointers relative to that one allocated block of memory. This is new functionality. We need a way to describe it. The OpenACC committee is considering new syntax to effect copying of data pointed to by a pointer, followed by translation of additional pointers relative to that copied data.

A third problem is that the data members are private to the class. Even if we have a solution to the first problem (applying directives to a class from outside the class definition), we can't refer to private data members. This is a somewhat more difficult problem, but a workaround is possible within a true C++ compiler. It may be more difficult for source-to-source translators, which typically do not perform a complete C++ parse and compile. All that said, the OpenACC committee is working to define the right functionality and performance for **`std::vector`** in particular, and C++ classes in general, for OpenACC 3.0. We expect this to take many more months, but we hope to have this finished sometime in 2015.

In the near term, PGI will implement a few changes in the pgc++ compiler that affect these examples. The most relevant has to do with class member pointers. In the **`devcopyin`** function in the last code example above, the **`enter data`** directive explicitly specifies both the **`this`** class and the member **`mydata`**. If the **`this`** reference is left out, only the data vector would be moved to the device, and references to **`a`** or **`b`** on the device would be invalid. In the future, a member reference in a data clause will cause the parent object to be allocated on the device as well. That's a change that will be relevant for C++ programmers in particular, especially when deleting those objects.

One last important warning: Look at the **`devcopyout`** function in that last code example. Note that the data vector **`mydata`** is copied from the device back to the host, but the **`myvector`** instance, represented by the **`this`** pointer, is deleted without updating the host. If you replace the **`delete`** clause with **`copyout`**, you will get some very strange behavior. That would copy the data members from the device to the host, including the device pointer in the device copy of **`mydata`**. Because this is a pointer to device memory, the host can't dereference it, and when the destructor for that object is eventually invoked, the **`delete []mydata`** will fail as well. When using classes with member pointers, beware of updating the host class from the device. This will be addressed in an upcoming version of the OpenACC specification as well, and PGI will implement that behavior when it is defined.

Closing Comments

Real C++ programs are syntactically related to C, but have much richer encapsulation and reuse behaviors. With OpenACC 2.0 dynamic data lifetimes (using **`enter data`** and **`exit data`** directives) and support for class member pointers on the accelerator device, C++ programmers can use OpenACC much more effectively. Parallel programming isn't easy, and heterogeneous programming adds complexity in data management, but the OpenACC features now available in pgc++ make these much more natural for C++ programmers.

PGI®