11 Tips for Maximizing Performance with PGI Directives in Fortran

List of Tricks

1. Privatize Arrays

2. Make While Loops Parallelizable

3. Rectangles Are Better Than Triangles

4. Restructure Linearized Arrays with Computed Indices

5. Privatize Live-out Scalars

6. Inline Function Calls in Directive Regions

7. Watch for Runtime Device Errors

8. Be Aware of Data Movement

9. Use Directive Clause to Optimize Performance

10. Use Data Regions to Avoid Inefficiencies

11. Leave Data on GPU Across Procedure Boundaries

Trick #1: Privatize Arrays

Some loops will fail to offload because parallelization is inhibited by arrays that must be privatized for correct parallel execution. In an iterative loop, data which is used only during a particular iteration can be declared private. And in general code regions, data which is used within the region but is not initialized prior to the region and is re-initialized prior to any use after the region can be declared private.

For example, if the following code is compiled:

```
!$acc region
   do i = 1, M
      do j = 1, N
         do jj = 1, 10
            tmp(jj) = jj
         end do
         A(i,j) = sum(tmp)
      enddo
   enddo
!$acc end region
```

Informational messages similar to the following will be produced:

```
% pgfortran -ta=nvidia -Minfo=accel private.f
privatearr:
      4, Generating copyout(a(1:m,1:n))
         Generating copyout(tmp(1:10))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
      5, Parallelization would require privatization of array 'tmp(1:10)'
      6, Parallelization would require privatization of array 'tmp(1:10)'
         Accelerator kernel generated
          5, !$acc do seq
          6, !$acc do seq
             Non-stride-1 accesses for array 'a'
```

A CUDA kernel is generated, but it will be very inefficient because it is sequential. If you further specify using a `loop` directive `private` clause that it is safe to privatize array `tmp` in the scope of the do j loop:

```
!$acc region
   do i = 1, M
!$acc do private(tmp)
      do j = 1, N
         do jj = 1, 10
            tmp(jj) = jj
         end do
         A(i,j) = sum(tmp)
      enddo
   enddo
!$acc end region
```

It will provide the PGI compiler with the information necessary to successfully compile the nested loop into a fully parallel CUDA kernel for execution on an NVIDIA GPU:

```
% pgfortran -ta=nvidia -Minfo=accel -V11.9 private2.f
privatearr:
      4, Generating copyout(a(1:m,1:n))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
      5, Loop is parallelizable
      7, Loop is parallelizable
         Accelerator kernel generated
          5, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
          7, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
             CC 1.0 : 9 registers; 56 shared, 8 constant,
                      0 local memory bytes; 100% occupancy
             CC 1.3 : 9 registers; 56 shared, 8 constant,
                      0 local memory bytes; 100% occupancy
             CC 2.0 : 19 registers; 8 shared, 64 constant,
                      0 local memory bytes; 100% occupancy
      8, Loop is parallelizable
     11, Loop is parallelizable
```

Note that the compiler will by default generate versions of the kernel that can be executed on CUDA devices with compute capability 1.1, 1,3 or 2.0. You can restrict code generation to a specific compute capability, say 2.0 for Fermi-class GPUs, using the compiler option -ta=nvidia:cc20.

Trick #2: Make While Loops Parallelizable

The PGI Accelerator compiler can't automatically convert while loops into a form suitable to run on the GPU. But it is often possible to manually convert a while loop into a countable rectangular do loop. For example, if the following code is compiled:

```
!$acc region
   i = 0
   do, while (.not.found)
      i = i + 1
      if (A(i) .eq. 102) then
         found = i
      endif
   enddo
!$acc end region
```

Informational messages similar to the following will be produced:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel while.f -c
PGF90-W-0155-Accelerator region ignored; see -Minfo messages  (while.f: 6)
while:
      6, Accelerator region ignored
      8, Accelerator restriction: invalid loop
  0 inform,   1 warnings,   0 severes, 0 fatal for while
```

But if the loop is restructured into the following form as a do loop:

```
!$acc region
   do i = 1, N
      if (A(i) .eq. 102) then
         found(i) = i
      else
         found(i) = 0
      endif
   enddo
!$acc end region
print *, 'Found at ', maxval(found)
```

It will provide the PGI compiler with the information necessary to successfully compile the nested loop for execution on an NVIDIA GPU:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel while2.f -c
while:
      5, Generating copyin(a(1:n))
         Generating copyout(found(1:n))
         Generating compute capability 2.0 binary
      6, Loop is parallelizable
         Accelerator kernel generated
          6, !$acc do parallel, vector(256) ! blockidx%x threadidx%x
             Using register for 'found'
             CC 2.0 : 8 registers; 4 shared, 60 constant,
                      0 local memory bytes; 100% occupancy
```

Trick #3: Rectangles Are Better Than Triangles

All loops must be rectangular. For triangular loops, the compiler will either serialize the inner loop or make the inner loop rectangular by adding an implicit if statement to skip the lower part of the triangle. For example, if the following triangular loop is compiled:

```
!$acc region
do i = 1, M
   do j = i, N
      A(i,j) = i+j
   enddo
enddo
!$acc end region
```

Here's the triangular loop!

Informational messages similar to the following will be produced:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel -c triangle.f
triangle:
      4, Generating copyout(a(1:m,:))
         Generating compute capability 2.0 binary
      5, Loop is parallelizable
         Accelerator kernel generated
          5, !$acc do parallel, vector(256) ! blockidx%x threadidx%x
             CC 2.0 : 21 registers; 12 shared, 60 constant,
                      0 local memory bytes; 83% occupancy
      6, Loop is parallelizable
```

While the loops seemed to have been parallelized, the resulting code will **likely fail**. Why? Because the compiler copies out the entire `A` array from device to host and in the process copies garbage values into the lower triangle of the host copy of `A`. However, if a `copy` clause is specified on the accelerator region boundary correct code will be generated. For example, after compiling the following loop:

```
!$acc region copy(A)
do i = 1, M
   do j = i, N
      A(i,j) = i+j
   enddo
enddo
!$acc end region
```

Informational messages similar to the following will be produced:

```
pgfortran -ta=nvidia:cc20 -Minfo=accel -c triangle2.f
triangle:
      4, Generating copy(a(:,:))
         Generating compute capability 2.0 binary
      5, Loop is parallelizable
         Accelerator kernel generated
          5, !$acc do parallel, vector(256) ! blockidx%x threadidx%x
             CC 2.0 : 21 registers; 12 shared, 60 constant,
                      0 local memory bytes; 83% occupancy
      6, Loop is parallelizable
```

Trick #4: Restructure Linearized Arrays with Computed Indices

It is not uncommon for legacy codes to use computed indices for computations on multi-dimensional arrays that have been linearized. For example, if the following loop with a computed index into the linearized array `A` is compiled:

```
!$acc region
   do i = 1, M
      do j = 1, N
         idx = ((i-1)*M)+j
         A(idx) = B(i,j)
      enddo
   enddo
!$acc end region
```

Informational messages similar to the following will be produced:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel linearization.f
linear:
      4, Generating copyout(a(:))
         Generating copyin(b(1:m,1:n))
         Generating compute capability 2.0 binary
      5, Parallelization would require privatization of array 'a(:)'
      6, Parallelization would require privatization of array 'a(:)'
         Accelerator kernel generated
          5, !$acc do seq
          6, !$acc do seq
             Non-stride-1 accesses for array 'b'
             CC 2.0 : 16 registers; 0 shared, 72 constant,
                      0 local memory bytes; 16% occupancy
```

The code will run on the GPU but it **will execute sequentially and run very slowly**. You have two options. First, the loop can be restructured to remove linearization:

```
!$acc region
   do i = 1, M
      do j = 1, N
         A(i,j) = B(i,j)
      enddo
   enddo
!$acc end region
```

Allowing the compiler to successfully generate a parallel GPU code:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel linearization2.f
linear:
      4, Generating copyout(a(1:m,1:n))
         Generating copyin(b(1:m,1:n))
         Generating compute capability 2.0 binary
      5, Loop is parallelizable
      6, Loop is parallelizable
         Accelerator kernel generated
          5, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
          6, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
             CC 2.0 : 11 registers; 16 shared, 64 constant,
                      0 local memory bytes; 100% occupancy
```

Or second, independent clauses can be specified on the do loops to provide the compiler with the information necessary to safely parallelize the loops:

```
!$acc region
!$acc do independent
   do i = 1, M
!$acc do independent
      do j = 1, N
         idx = ((i-1)*M)+j
         A(idx) = B(i,j)
      enddo
   enddo
!$acc end region
```

Trick #5: Privatize Live-out Scalars

It is common for loops to initialize scalar work variables, and for those variables to be referenced or re-used after the loop. Such a variable is called a “live out” scalar, because correct execution may depend on its having the last value it was assigned in a serial execution of the loop(s). For example, if the following loop with a live out variable `idx` is compiled:

```
!$acc region
      do i = 1, M
         do j = 1, N
            idx = i+j
            A(i,j) = idx
         enddo
      enddo
!$acc end region
      print *, idx, A(1,1), A(M,N)
```

Informational messages similar to the following will be produced:

```
% pgfortran liveout.f -ta=nvidia:cc13 -Minfo=accel -c
liveout:
      4, Generating copyout(a(1:m,1:n))
         Generating compute capability 1.3 binary
      5, Loop is parallelizable
      6, Inner sequential loop scheduled on accelerator
         Accelerator kernel generated
          5, !$acc do parallel, vector(256) ! blockidx%x threadidx%x
          6, !$acc do seq
             CC 1.3 : 9 registers; 48 shared, 16 constant,
                      0 local memory bytes; 100% occupancy
      7, Accelerator restriction: induction variable live-out from loop: idx
      8, Accelerator restriction: induction variable live-out from loop: idx
```

While some code will run on the GPU, the inner loop is executed sequentially. Looking at the code, the use of `idx` in the print statement is only for debugging purposes. In this case, you know the computations will still be valid even if `idx` is privatized so the code can be modified as follows:

```
!$acc region
      do i = 1, M
!$acc do private(idx)
         do j = 1, N
            idx = i+j
            A(i,j) = idx
         enddo
      enddo
!$acc end region
      print *, idx, A(1,1), A(M,N)
```

A much more efficient fully parallel kernel will be generated:

```
% pgfortran liveout2.f -ta=nvidia:cc13 -Minfo=accel -c
liveout:
      4, Generating copyout(a(1:m,1:n))
         Generating compute capability 1.3 binary
      5, Loop is parallelizable
      7, Loop is parallelizable
         Accelerator kernel generated
          5, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
          7, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
             CC 1.3 : 8 registers; 56 shared, 12 constant,
                      0 local memory bytes; 100% occupancy
```

Note that the value printed out for `idx` in the print statement will be different than in a sequential execution of the program.

Trick #6: Inline Function Calls in Directives Regions

One of the most common barriers to maximum GPU performance is the presence of function calls in the region. To run efficiently on the GPU, the compiler must be able to inline function calls. There are two ways to invoke automatic function inlining with the PGI Accelerator compilers:

1. If the function(s) to be inlined are in the same file as the section of code containing the accelerator region, you can use the `–Minline` compiler command-line option to enable automatic procedure inlining. This will enable automatic inlining of functions throughout the file, not only within the accelerator region. If you would like to restrict inlining to specific functions, say func1 and func2, use the option `–Minline=func1,func2`. To learn more about controlling inlining with `–Minline`, see the

`pgfortran` man page, or just type `pgfortran -help -Minline` in a shell window with the environment initialized for use of the PGI Accelerator compilers.

2. If the function(s) to be inlined are in a separate file from the code containing the accelerator region, you need to use the inter-procedural optimizer with automatic inlining enabled by specifying `-Mipa=inline` on the compiler command-line. `-Mipa` is both a compile-time and link-time option, so you need to specify it on the command-line when linking your program as well for inlining to occur. As with `-Minline`, you can learn more about controlling inter-procedural optimizations and inlining from the `pgfortran` man pages, or using `pgfortran -help -Mipa`.

In some cases when working with Fortran, procedures can only be inlined automatically by enabling array reshaping with `-Minline,reshape` or `-Mipa=inline,reshape.` For example when a 2D array is passed as an actual argument to a corresponding 1D array dummy argument.

There are several restrictions on automatic inlining. A Fortran subprogram will not be inlined if any of the following applies:

- It is referenced in a statement function.
- A common block mismatch exists; in other words, the caller must contain all common blocks specified in the callee, and elements of the common blocks must agree in name, order, and type (except that the caller's common block can have additional members appended to the end of the common block).
- An argument mismatch exists; in other words, the number and type (size) of actual and formal parameters must be equal.
- A name clash exists, such as a call to subroutine `xyz` in the extracted subprogram and a variable named `xyz` in the caller.

If you encounter these or any other restrictions that prevent automatic inlining of functions called in accelerator regions, the only alternative is to inline them manually.

Trick #7: Watch for Runtime Device Errors

Once you have successfully offloaded code in an accelerator region for execution on the GPU, you can still encounter errors at runtime due to common porting or coding errors that are not exposed by execution on the host CPU.

If you encounter the following error message when executing a program:

```
Call to cuMemcpyDtoH returned error 700: Launch failed
```

This typically occurs when the device kernel returns an execution error due to an out-of-bounds or other memory access violation. For example the following code will generate such an error:

```
!$acc region
      do i = 1, M
         do j = 1, N
            A(i,j) = B(i,j+1) << out-of-bounds
         enddo
      enddo
!$acc end region
```

The only way to isolate such errors currently is through inspection of the code in the accelerator region, or by compiling and executing on the host using the `-Mbounds` command-line option. This option will instrument the executable to print an error message for out-of-bounds array accesses.

If you encounter the following error message when executing a program:

```
Call to cuMemcpy2D returned error 1: Invalid value
```

This typically occurs if there is an error copying data to or from the device. For example, the following code will generate such an error:

```
      parameter(N=1024,M=512)
      real :: A(M,N), B(M,N)
      ...
!$acc region copyout(A), copyin(B(0:N,1:M+1))  <<< Bad bounds
      do i = 1, M                                   for copyin
         do j = 1, N
            A(i,j) = B(i,j+1)
         enddo
      enddo
!$acc end region
```

The only way to isolate such errors currently is through inspection of the code in the accelerator region or inspection of the `-Minfo` informational messages at compile time.

Trick #8: Be Aware of Data Movement

Having successfully offloaded a CUDA kernel using PGI Accelerator directives, you should understand and try to optimize data movement between host memory and GPU device memory.

You can see exactly what data movement is occurring for each generated CUDA kernel by looking at the informational messages emitted by the PGI Accelerator compiler:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel -c jacobi.f90
jacobi:
     18, Generating copyin(a(1:m,1:n))
         Generating copyout(a(2:m-1,2:n-1))
         Generating copyout(newa(2:m-1,2:n-1))
         ...
```

Array a being copied from host memory to GPU device memory before CUDA kernel launch

Elements of arrays a and newa copied back to host memory after CUDA kernel execution

You can see how much execution time is spent moving data between host memory and device memory by linking your executable with the `time` sub-option added to `-ta=nvidia` command-line option:

```
% pgfortran -ta=nvidia:time jacobi.f90
% a.out

<output from program>

Accelerator Kernel Timing data
  jacobi
```

79,825 microseconds spent executing kernels

```
    18: region entered 798 times
        time(us): total=5575112 init=4565273 region=1009839
                  kernels=79825 data=385751
        w/o init: total=1009839 max=12347 min=1191 avg=1265
        20: kernel launched 798 times
            grid: [16x16]  block: [16x16]
            time(us): total=47315 max=70 min=58 av
        24: kernel launched 798 times
            grid: [1]  block: [256]
            time(us): total=9067 max=13 min=11 avg=11
        27: kernel launched 798 times
            grid: [16x16]  block: [16x16]
            time(us): total=23443 max=35 min=28 avg=29
```

385,751 microseconds spent on moving data between host memory and GPU device

Once you have examined and timed the data movement required at accelerator region boundaries, there are several techniques you can use to minimize and optimize data movement.

Trick #9: Use Directive Clauses to Optimize Performance

By default, the PGI Accelerator compilers will move the minimum amount of data required to perform the necessary computations on the GPU. For example, if the following code is compiled:

```
      change = tolerance + 1 ! get into the while loop
      iters = 0
      do while ( change > tolerance )
         iters = iters + 1
         change = 0
!$acc region
         do j = 2, n-1
            do i = 2, m-1
               newa(i,j) = w0 * a(i,j) + &
               w1 * (a(i-1,j) + a(i,j-1) + a(i+1,j) + a(i,j+1) ) + &
               w2 * (a(i-1,j-1) + a(i-1,j+1) + a(i+1,j-1) + a(i+1,j+1) )
               change = max( change, abs( newa(i,j) - a(i,j) ) )
            enddo
         enddo
         a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
!$acc end region
      enddo
```

Feedback messages similar to the following will be produced:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel -c jacobi.f90
jacobi:
     18, Generating copyin(a(1:m,1:n))
         Generating copyout(a(2:m-1,2:n-1))
         Generating copyout(newa(2:m-1,2:n-1))
         Generating compute capability 2.0 binary
     19, Loop is parallelizable
     20, Loop is parallelizable
         Accelerator kernel generated
         19, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
         20, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
             Cached references to size [18x18] block of 'a'
             CC 2.0 : 18 registers; 1328 shared, 104 constant,
                      0 local memory bytes; 100% occupancy
         24, Max reduction generated for change
     27, Loop is parallelizable
         Accelerator kernel generated
         27, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
             !$acc do parallel, vector(16) ! blockidx%y threadidx%y
             CC 2.0 : 10 registers; 16 shared, 80 constant,
                      0 local memory bytes; 100% occupancy
```

Some things to note:

- Only the interior elements of the arrays `a` and `newa` are modified, so only those elements are copied out of the GPU memory to host memory
- Performance degrades dramatically because data being copied is not contiguous and is using small transfers
- Array `newa` is just a temporary array which does not need to be initialized before kernel execution and is not used after kernel execution.

If we modify the code as follows by adding clauses to the `acc region` directive to specify that the entire array `a` should be copied in and out, and that the array `newa` can be treated as GPU-local (i.e. as a scratch array that does not need to be copied):

```
      change = tolerance + 1 ! get into the while loop
      iters = 0
      do while ( change > tolerance )
         iters = iters + 1
         change = 0
!$acc region copy(a) local(newa)
         do j = 2, n-1
            do i = 2, m-1
               newa(i,j) = w0 * a(i,j) + &
               w1 * (a(i-1,j) + a(i,j-1) + a(i+1,j) + a(i,j+1) ) + &
               w2 * (a(i-1,j-1) + a(i-1,j+1) + a(i+1,j-1) + a(i+1,j+1) )
               change = max( change, abs( newa(i,j) - a(i,j) ) )
            enddo
         enddo
         a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
!$acc end region
      enddo
```

When re-compiled the PGI compiler emits the following feedback messages:

```
% pgfortran -ta=nvidia:cc20 -Minfo=accel -c jacobi2.f90
jacobi:
     18, Generating copy(a(:,:))
         Generating local(newa(:,:))
         Generating compute capability 2.0 binary
     ...
```

The copy of array `a` will be much more efficient, and data movement for array `newa` has been completely eliminated.

Trick #10: Use Data Regions to Avoid Inefficiencies

The PGI Accelerator programming model has two kinds of regions: a *compute region* and a *data region*. In a compute region, loops to be executed on the GPU are delineated using the `!$acc region` and `!$acc end region` directives. For example, we might enclose a weighted five-point stencil operation in Fortran as:

```
!$acc region
    do i = 2, n-1
      do j = 2, m-1
        b(i,j) = 0.25*w(i)*(a(i-1,j)+a(i,j-1)+ &
                            a(i+1,j)+a(i,j+1)) &
                +(1.0-w(i))*a(i,j)
      enddo
    enddo
!$acc end region
```

When compiled, the PGI Accelerator compiler emits the following messages:

```
s1:
      7, Generating copyin(w(2:n-1))
         Generating copyin(a(1:n,1:m))
         Generating copyout(b(2:n-1,2:m-1))
      8, Loop is parallelizable
      9, Loop is parallelizable
         Accelerator kernel generated
          8, !$acc do parallel, vector(16)
             Cached references to size [16] block of 'w'
          9, !$acc do parallel, vector(16)
             Cached references to size [18x18] block of 'a'
```

Compiler is sending a portion of array w and array a to the GPU, and copying modified portion of array b back to host memory

Shows parallelism schedule: Loops are executed in 16x16 blocks.

However, there is a serious problem with this particular program. The loop does roughly 8*n*m operations, **but transfers roughly 8*n*m bytes to do it.** The data transfer between the host and the GPU will dominate any performance advantage gained from the parallelism on the GPU. We certainly don't want to send data between the host and GPU for each iteration. Enter the data region.

A data region looks similar to a compute region, but defines only data movement between host memory and GPU device memory. In an iterative solver, a data region can be placed outside the iteration loop, and an enclosed compute region around the computational kernel. A more complete example might look as follows:

```
!$acc data region copy(a(1:n,1:m)) local(b(2:n-1,2:m-1)) copyin(w(2:n-1))
      do while(...)
!$acc region
         do i = 2, n-1
            do j = 2, m-1
               b(i,j) = 0.25*w(i)*(a(i-1,j)+a(i,j-1)+ &
                                   a(i+1,j)+a(i,j+1)) &
                      +(1.0-w(i))*a(i,j)
            enddo
         enddo
         do i = 2, n-1
            do j = 2, m-1
```

```
               a(i,j) = b(i,j)
            enddo
        enddo
!$acc end region
     enddo
!$acc end data region
```

Now, any input data is copied to GPU device memory at entry to the data region, and the results copied back to host memory at exit of the data region. **Inside the while loop, there is essentially no data movement between the host and the GPU.** This will run several times faster than the original program.

A data region will typically contain one or more compute regions; data used in a compute region that was moved to the accelerator in an enclosing data region directive are not moved at the boundaries of the compute region.

The `updatein` and `updateout` clauses allow fine-tuning of data movement at region boundaries. You can add an `updatein` clause to a compute region directive for data that was allocated on the GPU in an enclosing data region, but which has been updated on the host between the beginning of the data region and the beginning of the compute region. This tells the compiler the arrays or parts of arrays that need to be copied from host memory to GPU device memory at entry to the compute region. Similarly, you would add an `updateout` clause to a compute region when you have data allocated on the GPU in an enclosing data region, but you want some or all of the host copy of that array updated at the exit of the compute region.

Using data regions, it is often possible to substantially reduce the amount of data movement in program units that include multiple accelerator compute regions.

Trick #11: Leave Data on GPU Across Procedure Boundaries

`Mirror` Directive and Clause

Data regions enable the programmer to leave data in GPU device memory and re-use it across multiple procedures. For this feature, two additional directives are provided.

The PGI Accelerator Fortran `mirror` directive applies to Fortran allocatable arrays. This directive informs the compiler that allocate and deallocate statements for this array should allocate copies both in host memory *and* in the GPU device memory. When mirrored arrays appear in host code, the host copy is used; when they appear in PGI Accelerator compute regions, the GPU copy is used. Using the `mirror` directive with module allocatable arrays gives them global visibility. For example:

```
      module glob
      real, dimension(:), allocatable :: x
!$acc mirror( x )
      end glob

      subroutine sub( y )
      use glob
      real, dimension(:) :: y
!$acc region
```

```
      do i = 1, ubound(y,1)
         y(i) = y(i) + x(i)
      enddo
!$acc end region
      end subroutine
```

In this example, when array `x` is allocated, a copy on the GPU will be allocated as well. When the accelerator region in subroutine sub is executed, the region will use the values in the GPU copy of `x` with no data movement at the accelerator region boundary.

The `update` directive can be used to synchronize data between the GPU and host copies of the array. The following example updates a sub-array of the host copy of `x` from GPU data:

```
     subroutine sync
     use glob
!$acc update host( x(2:499) )
     end subroutine
```

In addition to the `mirror` directive, there is a `mirror` clause for use in data regions; the `mirror` clause is like the `mirror` directive, except the GPU copy only has the lifetime of the data region on which the `mirror` clause occurs. Here are some things to note:

- When the data region is entered, if an array in a `mirror` clause is allocated on the host (or is not allocatable), it will be allocated on the GPU with the same size.
- If an allocatable array in a `mirror` clause is not allocated, a GPU copy will not be allocated.
- If an allocatable array in a `mirror` clause is allocated or deallocated in the data region, the GPU copy will likewise be allocated or deallocated. It's important to note that the allocation or deallocation does not imply any data movement.
- If the host data needs to be copied to the GPU at the region boundary for correct execution, either a `copyin` clause should be used instead of `mirror` or explicit `update` directives must be added as well.

`Mirror` directives and Fortran modules enable use of global device-resident data across procedure boundaries, but what if you need to pass a variable with a device-resident copy to a function or subroutine?

`Reflected` Directive

Procedure arguments with device-resident copies must be passed using the PGI Accelerator Fortran `reflected` directive, which applies to dummy arguments. This directive informs the compiler that the specified array dummy arguments appear in a data region clause in the caller, or are mirrored on the GPU. Consider the previous example expanded as follows:

```
      module glob
      real, dimension(:), allocatable :: x
!$acc mirror( x )
      contains
         subroutine sub( y )
         real, dimension(:) :: y
!$acc reflected(y)
!$acc region
         do i = 1, ubound(y,1)
            y(i) = y(i) + x(i)
         enddo
!$acc end region
         end subroutine
      end module

      subroutine roo( z )
      use glob
      real :: z(:)
!$acc data region copy(z)
      call sub( z )
!$acc end data region
      end subroutine
```

The subroutine sub is contained within a module, and the dummy array `y` has the reflected attribute. The caller roo uses the module, making the interface to sub *explicit*; alternatively, sub can be an external subroutine but in that case it must have an interface block in roo. At the call site, the compiler knows that the host array `z` has a copy in GPU device memory, and that the subroutine sub needs both the host address and the GPU address for `z`. Within the subroutine sub, the compiler knows, because of the `reflected` directive, that the dummy argument `y` must have been copied to the GPU by the caller, and so the compute region in the subroutine incurs no data movement for either `y` (because it's reflected) or `x` (because it's mirrored).

The `reflected` directive only applies to dummy argument arrays, and can only be used when the procedure interface is *explicit*, in Fortran terms. This means the subprogram must appear in a module, and the caller in the same module, or in a scope where the module has been USEd, or the caller must have a matching interface block to the subprogram with the reflected directive. Another restriction in the current implementation is the whole array must be copied to the GPU to use the `reflected` directive.

Using data regions along with the `mirror` and `reflected` directives, it is possible to allocate and use data in GPU device memory across large portions of an application while minimizing the number of data transfers that must occur to keep the host and device copies coherent.