True OpenACC Deep Copy Beta Feature in PGI 18.7 Compilers

by Michael Wolfe, PGI Compiler Engineer

This article will describe a new feature in the PGI OpenACC compilers for GPU computing to support *deep copy* for composite data types (C struct, Fortran derived type, C++ struct and class). The compilers already support *manual deep copy* (/blogs/posts/deep-copy.htm), and the PGI Fortran compilers already support *automatic full deep copy*. Now, with PGI 18.7, we are delivering the first implementation of *true deep copy*, which allows the flexibility of manual deep copy with less work by the programmer.

Simple Example

Let's start with a very simple example. Here we show the same example in C and in Fortran, with a single struct and three dynamically-allocated members.

Simple Example in C

Imagine a C struct with two or more array members.

```
/* struct with array of floats and array of ints */
typedef struct{
  int n;
  float *a, *b;
  int *x;
}s1;
...
s1 g;
g.a = (float*)malloc(...);
g.b = (float*)malloc(...);
g.x = (int*)malloc(...);
...
for (i = 0; i < g.n; ++i)
  g.a[i] = g.b[g.x[i]];
```

If we want to port this parallel loop using OpenACC to the GPU, we have several options.

- One is to rewrite the loop by copying the struct pointer members to pointer variables and using the pointer variables in the loop and associated OpenACC data clauses.
- Another is to use *managed memory* by compiling with the `-ta=tesla:managed` command line option, supported by the PGI compilers since release 15.9, which changes the allocations to use CUDA Unified memory and lets the NVIDIA GPU device driver move data between system and device memory as needed.
- A third option is *manual deep copy*, introduced in OpenACC 2.6 and available in PGI compilers since early in 2018 (and now proposed for the OpenMP 5.0). Manual deep copy requires adding a directive or data clause for each dynamically-allocated member.

With PGI 18.7, we introduce *true deep copy*. With true deep copy, you insert extra directives in the struct definition to define properties of the struct members, such as the shape of the dynamic members and which of them should be copied with the struct. The extra directives are only inserted once in the struct definition; each variable declared with that struct type will inherit those properties. When the struct variable appears in a data clause, the OpenACC compiler and runtime will cooperate in appropriately copying the whole data structure, without you having to list each member. In this example, we add `shape` directives to declare the shape of the three member pointers. When the compiler sees the `copy(g)` data clause, it knows to also apply the `copy` behavior to each of the member pointers.

```
typedef struct{
  int n;
  float *a, *b;
  int *x;
  #pragma acc shape(a[0:n],b[0:n],x[0:n])
}s1;
...
#pragma acc parallel loop copy(g)
for (i = 0; i < g.n; ++i)
  g.a[i] = g.b[g.x[i]];
```

You can see the effect of the deep copy behavior by running such a program with the environment variable `PGI_ACC_NOTIFY=3` , which prints a line for each kernel launch (bit 1) and each data movement (bit 2). In this program, it will show the data upload to the GPU memory for the struct `g` followed by the upload for each member array before the parallel loop, then the download of each member array and the struct after the parallel loop.

In this example, two of those members are read-only, so they only need to be copied in. With the manual deep copy directives, we can optimize for this by using different data clauses for the two data members that don't need to be copied out. We can get the same behavior with *true deep copy* by adding a `policy` directive.

```
typedef struct{
  int n;
  float *a, *b;
  int *x;
  #pragma acc shape(a[0:n],b[0:n],x[0:n])
  #pragma acc policy<ainout> copy(a) copyin(b,x)
}s1;
...
#pragma acc parallel loop copy(g<ainout>)
for (i = 0; i < g.n; ++i)
  g.a[i] = g.b[g.x[i]];
```

The policy name `ainout` is specified on the data clause for the struct variable, and the data movement options from the policy are applied to the members. We will see an example with multiple named policies, to have different behavior in different parts of the application.

This feature has been under active discussion by the OpenACC committee for several years. It is now available for beta-testing, in PGI 18.7, with the command line option `-ta=tesla:deepcopy`. Depending on experiences of and feedback from our user community, we will adjust the syntax and behavior before finalizing a proposal for adoption in a future version of the OpenACC specification. in a future version of OpenACC.

Simple Example in Fortran

Let's look at the same example in Fortran, to highlight the differences in that language. Imagine a Fortran derived type with two or more allocatable array members:

```
type d1
  integer :: n
  real, allocatable :: a(:), b(:)
  integer, allocatable :: x(:)
end type
...
type(dt) :: g
...
allocate(g%a(...), g%b(...), g%x(...))
...
do i = 1, g%n
  g%a(i) = g%b( g%x(i) )
enddo
```

When porting this parallel loop using OpenACC to the GPU, we also have the same options as in C.

- Rewriting the loop to use array pointer variables instead of the allocatable struct members is more problematic in Fortran, because the allocatable members would have to have the `target` attribute, so it's less useful.
- Using *managed memory* option works with Fortran just as it does with C, again with the `-ta=tesla:managed` build option.
- The *manual deep copy* directives have been used very successfully with Fortran, though with the same programming overhead as in C.

With PGI 18.7, the `-ta=tesla:deepcopy` option also enables *true deep copy* directives for Fortran. The `shape` directive isn't needed much for Fortran, but the `policy` directive can still be useful. In this example, we create a policy to optimize data movement, as we did with the C example.

```
type d1
  integer :: n
  real, allocatable, target :: a(:), b(:)
  integer, allocatable, target :: x(:)
  !$acc policy<ainout> copy(a) copyin(b,x)
end type
...
!$acc parallel loop copy(g<ainout>)
do i = 1, g%n
  g%a(i) = g%b( g%x(i) )
enddo
```

With this policy, the struct `g` will be allocated and initialized when entering the parallel loop, then each of the allocatable members will be allocated and copied in as well. At the end of the loop, the `a` member and the struct `g` will be copied out, while `b` and `x` will be simply deallocated.

Slightly More Complex Example

That was a simple example, with a simple structure `g` that had three allocatable members. What if `g` were itself an array, and each array element had the same allocatable members. Here we show each of the various options with an array of struct or derived type, which highlights the value of the new deep copy behavior.

More Complex Example in C

Now imagine that we have an array of structs. With the new *true deep copy* behavior, we can use the same `shape` directive in the struct definition, and the compiler will apply that for each element of the `g` array.

```
/* struct with array of floats and array of ints */
typedef struct{
  int n;
  float *a, *b;
  int *x;
  #pragma acc shape(a[0:n],b[0:n],x[0:n])
  #pragma acc policy<ainout> copy(a) copyin(b,x)
}s1;
...
s1* g;
g = (s1*)malloc(m*sizeof(s1));
for (j = 0; j < m; ++j) {
  g[j].n = ...;
  g[j].a = (float*)malloc(...);
  g[j].b = (float*)malloc(...);
  g[j].x = (float*)malloc(...);
}
...
#pragma acc parallel loop copy(g<ainout>[0:m])
for (j = 0; j < m; ++j) {
  for (i = 0; i < g[j].n; ++i)
    g[j].a[i] = g[j].b[g.x[i]];
}
...
```

Now the compiler and runtime will apply the policy and shape information to each element of the struct array. You can get the same behavior and level of control with true deep copy that you had with manual deep copy, but without so much programming overhead. Compare the code above with the same program written with manual deep copy:

```
...
#pragma acc enter data copyin(g[0:m])
for (j = 0; j < m; ++j) {
  #pragma acc enter data copyin(g[j].a[0:g[j].n])
  #pragma acc enter data copyin(g[j].b[0:g[j].n])
  #pragma acc enter data copyin(g[j].x[0:g[j].n])
}
#pragma acc parallel loop present(g)
for (j = 0; j < m; ++j) {
  for (i = 0; i < g[j].n; ++i)
    g[j].a[i] = g[j].b[g.x[i]];
}
for (j = 0; j < m; ++j) {
  #pragma acc exit data copyout(g[j].a[0:g[j].n])
  #pragma acc exit data delete(g[j].b[0:g[j].n])
  #pragma acc exit data delete(g[j].x[0:g[j].n])
}
#pragma acc exit data delete(g)
```

The extra user code to manage struct members in the manual deep copy version is handled in the OpenACC runtime with true deep copy.

More Complex Example in Fortran

Let's look at the more complex example in Fortran, with an array of derived type.

```
type d1
  integer :: n
  real, allocatable :: a(:), b(:)
  integer, allocatable :: x(:)
  !$acc policy<ainout> copy(a) copyin(b,x)
end type
...
type(dt), allocatable :: g(:)
...
allocate(g(...))
do j = 1, m
  allocate(g(j)%a(...), g(j)%b(...), g(j)%x(...))
enddo
...
!$acc parallel loop copy(g)
do j = 1, m
  do i = 1, g(j)%n
    g(j)%a(i) = g(j)%b( g(j)%x(i) )
  enddo
enddo
```

Using PGI 18.7 with the *true deep copy* behavior and the `policy` in the derived type definition, we can get the behavior of the manual deep copy version without the programming overhead. Note the same derived type definition for both the simple and more complex example. Again, we could have used *manual deep copy*, but as with C, it requires more programming effort:

```
!$acc enter data copyin(g)
do j = 1, m
  !$acc enter data copyin(g(j)%a, g(j)%b, g(j)%x)
enddo
...
!$acc parallel loop present(g)
do j = 1, m
  do i = 1, g(j)%n
    g(j)%a(i) = g(j)%b( g(j)%x(i) )
  enddo
enddo
...
do j = 1, m
  !$acc exit data copyout(g(j)%a) delete(g(j)%b, g(j)%x)
enddo
!$acc exit data delete(g)
```

True Deep Copy Directives

We have already seen an example of the `shape` directive in C. The `shape` directive adds information to C pointers to make them almost equivalent to Fortran allocatable and pointer members, giving the compiler shape information.

We will also show the `policy` directive. When doing a *manual deep copy*, you have full control over which members get allocated or copied in which direction. The `policy` directive gives you that control with true deep copy for data and for update directives.

Shape Directive

The `shape` directive is mostly for C programs, to associate a shape (length) to a pointer member of a derived type. The simplest form is the keyword `shape` followed by a parenthesized, comma-separated list of member names with shape information, as we saw in the examples above:

```
    typedef struct{
      int n;
      float *a, *b;
      int *x;
      #pragma acc shape(a[0:n], b[0:n])
    }s1;
```

The shape for C programs will be a start position (almost always zero) and a length. These may be simple expressions, but the components of the expression must be integer constants, integer global variables, or integer members of the same struct type. By default, when deep copy is enabled, unless a policy is invoked, all shaped members will participate in the data or update operation. You can have multiple shape directives in the same type declaration, but only one shape for each member, unless you use named shapes (below).

init_needed clause:
In our example datatype, there is one member, `n`, that is used to describe the shape of the members. We want that member to always be initialized, even if the struct variable itself appears in a `create` data clause, which would not copy any data members. To enforce initialization of such members, the `shape` directive can have an `init_needed` clause, telling the compiler that some member needs to always be initialized.

```
    typedef struct{
      int n;
      float *a, *b;
      int *x;
      #pragma acc shape(a[0:n], b[0:n]) init_needed(n)
    }s1;
```

Shape names:
The shape may itself be named. This allows for the rare case where member arrays may have shape information stored in different expressions. We will see how the shape name is used in a policy directive below.

```
    typedef struct{
      int n;
      float *a, *b;
      int *x;
      #pragma acc shape<s1shape>(a[0:n], b[0:n]) init_needed(n)
    }s1;
```

relative pointers:
In C programs, we may have two or more member pointers that point to the same block of memory. For instance, there may be a pointer to the start of a vector, a pointer to the current position in the vector, and a pointer to the end of the vector or, more commonly, just past the end of the vector.

```
    typedef struct{
      float *start, *curr, *end;
    }v1;
    v1 x;
    for (float* p = x.start; p < x.end; ++p) ...
```

When copying this structure to the device, we don't want the compiler to treat these like three independent pointers to three arrays. Moreover, `x.end` isn't really a valid pointer; dereferencing `x.end` will result either in an invalid reference, or it will point into some other block of memory. What we want is for `x.curr` and `x.end` to be set in device memory at the same relative position to `x.start`. This is accomplished with the `relative` clause.

```
    typedef struct{
      float *start, *curr, *end;
      #pragma acc shape(start[0:n]) relative(start:curr,end)
    }v1;
```

Policy Directive

The default deep copy *policy* is to apply the data clause behavior to each *shaped* member in a struct or derived type variable. This is used in any `data`, `enter data`, `exit data`, or `update` directive. In Fortran, all allocatable or pointer members are *shaped*, because the array shape is part of the array descriptor stored in the derived type. In C, we use the `shape` directive to provide shape information for each pointer member.

Simple Policies:
The `policy` directive is used to create one or more named behaviors to tune data movement. It's useful in C and Fortran both. The simplest form is the `policy` keyword with a policy name in angle brackets (`<policyname>`), followed by data clauses containing member names.

The clauses supported are the normal data clauses (`copy`, `copyin`, `copyout`, `create`, `no_create`, `deviceptr`, or the) or the `update` clause. The data clause can include a shape description for any member as well, which overrides the default shape for that member when this policy is invoked.

```
    typedef struct{
      int n;
      float *a, *b;
      int *x;
      #pragma acc shape(a[0:n], b[0:n]) init_needed(n)
      #pragma acc policy<ainout> copy(a) copyin(b, x[0:n])
    }s1;
    s1* g;
```

A named policy gets invoked by including the policy name in angle brackets after a variable name in a data clause or update directive clause.

```
    #pragma acc enter data copy(g<ainout>[0:m])
```

Excluding Members:
As mentioned the default behavior for data directives is to always process any shaped member. Even when using a named shape, a shaped member that is not named on the policy directive will be processed according to the data clause on which the struct variable name appears. For instance, with the declaration:

```
    typedef struct{
      int n;
      float *a, *b;
      int *x;
      #pragma acc shape(a[0:n], b[0:n]) init_needed(n)
      #pragma acc policy<p1> copyin(b)
    }s1;
    s1* g;
```

In the following data directive:

```
    #pragma acc data copy(g<p1>[0:m])
```

The pointer members `a` and `b` will both be processed for each element of `g`, because `a` appears in a `shape` directive as well, so it is shaped. The pointer member `x` will not be processed, because it is not shaped. We can create a policy that will copy only the `b` member by changing the default behavior for that policy from *include* to *exclude*, that is, to exclude any members not named on the `policy` directive:

```
typedef struct{
  int n;
  float *a, *b;
  int *x;
  #pragma acc shape(a[0:n], b[0:n]) init_needed(n)
  #pragma acc policy<p1> copyin(b)
  #pragma acc policy<p2> copyin(b) default(exclude)
}s1;
s1* g;
```

In the following data directive:

```
#pragma acc data copy(g<p2>[0:m])
```

only the pointer member `b` will be processed for each element of `g`; the member `a` will be excluded, even though it is shaped.

Nested Policies:
In even more complex cases, the pointer member is itself a pointer to a struct type or derived type. In that case, the member datatype may have its own policies. To support that, the member policy name may appear in the data clause of a policy:

```
typedef struct{
  long ng;
  s1* mg;
  #pragma acc shape(mg[0:ng]) init_needed(ng)
  #pragma acc policy<q1> copyin(mg<p1>)
}s2;
```

Update Policies:
When using a composite data type in an `update` directive with deep copy enabled, the default behavior is to update all shaped members. You can override this behavior with an update policy, that is, a policy with `update` movement clauses. For instance, we can extend our example above with two update policies:

```
typedef struct{
  int n;
  float *a, *b;
  int *x;
  #pragma acc shape(a[0:n], b[0:n]) init_needed(n)
  #pragma acc policy<p1> copyin(b)
  #pragma acc policy<p2> copyin(b) default(exclude)
  #pragma acc policy<u1> update(a)
  #pragma acc policy<u2> update(b)
}s1;
s1* g;
```

The following update directive:

```
#pragma acc update device(g<u1>[0:m])
```

will update only the `a` members of each element of `g`. Unlike with data policies, the default for an update policy is to *exclude* any unnamed members. If you want to include all shaped members for a named policy, add a `default(include)` clause to the update policy directive.

Recursive Data Types:
In this release, recursive data types, such as linked lists, trees, or any data type that has a pointer directly or indirectly the same data type, is not supported. We will depend on user feedback to prioritize how important this might be.

C++ Support:
In this release, deep copy support in C++ is equivalent to that in C. What is still missing is enough functionality to add deep copy directives to support, say, the `std::vector` and similar classes. These are still on the drawing board.

Summary

The new deep copy facility in the PGI OpenACC compilers greatly simplifies the management of complex dynamic data structures between system and device memory. This beta implementation has some limitations which will be addressed in the future. We depend on your feedback to improve the deep copy feature, and we will work with the OpenACC committee to add it to the specification.