Deep Copy in OpenACC: Support for Dynamically Nested Data Structures

by Michael Wolfe, PGI Compiler Engineer

One missing piece in OpenACC is support for arrays or structures that contain pointers to other arrays or structures. As larger programs are ported to GPUs using OpenACC, this missing piece has become a more serious problem. We're glad to report that OpenACC 2.6 has adopted the first step to a solution for this problem, what we call Manual Deep Copy. This feature allows a program to move a dynamically nested data structure between system memory and device memory, and PGI already has this partly implemented in the 17.10 version.

What follows is two sections, one for C++ (and C) programmers and a second for Fortran programmers. The solutions are essentially identical, but there are some language-specific issues that we want to call out.

Deep Copy in OpenACC for C++ Programmers

OpenACC was initially designed to port computational kernels from a host to an accelerator device, where the kernels operate on arrays or vectors of data. Many modern programs have much more complex data structures, such as a struct or class which contains pointers to other structures. Some of these other structures may themselves be structs or an arrays of struct, where the struct has other pointers to yet other structs or arrays. As a trivial example, think of the `std::vector` class. This is implemented as a class with a pointer to the vector data (actually three pointers, but that's beyond the scope of this section).

One solution for this problem with NVIDIA Tesla GPUs has been to use managed memory, enabled in the PGI compilers with the `-ta=tesla:managed` compiler flag on the compile and link steps. With this flag, all dynamically allocated data (with new or malloc) gets allocated in managed memory (CUDA Unified Memory), which gets paged on demand between system and device memory by the NVIDIA GPU device driver, and the addresses are the same in the two memories. In many cases, this is a sufficient solution. However, there can be downsides to using managed memory everywhere. First, the cost of a managed memory allocate operation is more expensive than a simple malloc. Also, data is moved between the memories on demand; there's no support for prefetching data ahead of time, or overlapping data movement with computation. If managed memory is not sufficient, then you need to be able to explicitly control data movement between the memories. With dynamic data structures, you need the pointers in device memory to point to the data in device memory.

Until OpenACC 2.6, there was no standard way to copy such a dynamic data structure to device memory and fix the pointers in device memory. PGI had implemented a first step to solve this problem in the 2017 releases, but it wasn't part of the specification and there were some missing features. As PGI 17.10 and subsequent releases implement more of OpenACC 2.6, we have made important steps to support dynamic data structures.

Attaching Pointers on the Device

Let's start with a first simple example. Suppose I have a class with pointers to three data arrays, each of size `n`:

```
    typedef class{
        float *x, *y, *z;
        float coefx, coefy, coefz;
        size_t n;
    }points1;
```

Now suppose I've initialized this structure in system memory, and now I want to copy this to and process this on the device. With OpenACC 2.5, the solution was to flatten the structure:

```
points1 p1;
...
float *px = p1.x;
float *py = p1.y;
float *pz = p1.z;
size_t pn = p1.n;
#pragma acc enter data copyin(px[0:pn], py[0:pn], pz[0:pn])
#pragma acc parallel loop default(present) ...
for (i = 0; i < pn; ++i) {
    px[i] = ....
}
```

While this works, it requires rewriting the loops that refer to `p1.x[i]` so that they instead use `px[i]`, and so on. It also doesn't scale, if the member arrays `x`, `y`, or `z` are themselves structs that contain other pointers.

With OpenACC 2.6, we can write this to refer to the struct or class members directly on the device, as long as the pointers in device memory get replaced to point to the device data. We call this the attach operation, to set a pointer on the device to point to the corresponding device data for the host pointer. There are several ways to do this. One way is to copy each of the members and the base struct, then to manually attach the pointers using the `acc_attach` API routine:

```
points1 p1;
...
#pragma acc enter data copyin(p1.x[0:p1.n], p1.y[0:p1.n], p1.z[0:p1.n])
#pragma acc enter data copyin(p1)
acc_attach(&p1.x);
acc_attach(&p1.y);
acc_attach(&p1.z);
#pragma acc parallel loop default(present) ...
for (i = 0; i < p1.n; ++i) {
    p1.x[i] = ....
}
```

The first `enter data` directive copies each of the data arrays to the device. The second `enter data` directive copies the base struct; this copies the struct directly from system memory, so the `p1.x`, `p1.y`, and `p1.z` pointers still contain host addresses. The `acc_attach` API routine is passed the address of the pointer to be attached. If the pointer is present on the device as well as the pointer target, the routine replaces the host pointer with the corresponding device pointer in device memory.

A second way to do this is to use the new OpenACC 2.6 attach clause; I emphasize that this clause will be available in the PGI compilers early in 2018:

```
points1 p1;
...
#pragma acc enter data copyin(p1.x[0:p1.n], p1.y[0:p1.n], p1.z[0:p1.n])
#pragma acc enter data copyin(p1) attach(p1.x, p1.y, p1.z)
#pragma acc parallel loop default(present) ...
for (i = 0; i < p1.n; ++i) {
    p1.x[i] = ....
}
```

A third way, and simpler to do this uses the new OpenACC 2.6 implicit attach behavior; this is available in the PGI 17.10 compilers:

```
points1 p1;
...
#pragma acc enter data copyin(p1)
#pragma acc enter data copyin(p1.x[0:p1.n], p1.y[0:p1.n], p1.z[0:p1.n])
#pragma acc parallel loop default(present) ...
for (i = 0; i < p1.n; ++i) {
    p1.x[i] = ....
}
```

The difference between this version and the previous is the base structure is copied to device memory first. When the member pointers are copied in the second enter data directive, the implementation will test if the pointers themselves (`p1.x`, `p1.y`, `p1.z`) are also present. In this case, they are present because of the first `enter data` directive. Because the pointers are present, they are implicitly attached to the `x`, `y`, and `z` device data.

Detaching Pointers on the Device

When done with the processing on the device, the data may need to be copied back to system memory, or at least deallocated. This is done with matchine `exit data` directives with either `copyout` or `delete` directives. One potential problem is to make sure the pointers in the device copy of the base structure are detached (that is, restored to the original host pointer values) before the base struct gets copied back to the host. As with the attach operation, there are three ways to do the detach operation. The first is to manually detach the pointers using the `acc_detach` API routine:

```
acc_detach(&p1.x);
acc_detach(&p1.y);
acc_detach(&p1.z);
#pragma acc exit data copyout(p1.x[0:p1.n],p1.y[0:p1.n],p1.z[0:p1.n])
#pragma acc exit data copyout(p1)
```

The `acc_detach` API routine has the same interface as acc_attach. If that pointer is present on the device and attached to a device address, it will be detached and restored to the host pointer value. Of course, if the base struct, `p1` here, is simply deleted, the detach operation isn't necessary.

A second way to do the same operations is to use the new OpenACC 2.6 detach clause:

```
#pragma acc enter data detach(p1.x, p1.y, p1.z)
#pragma acc enter data copyout(p1.x[0:p1.n], p1.y[0:p1.n], p1.z[0:p1.n])
#pragma acc enter data copyout(p1)
```

A third, simpler way uses the implicit detach behavior, again available in the 17.10 release:

```
#pragma acc enter data copyout(p1.x[0:p1.n], p1.y[0:p1.n], p1.z[0:p1.n])
#pragma acc enter data copyout(p1)
```

It is important that the member pointers are processed before the base structure, so the pointers in the base structure can be detached first.

Array of Struct or Class

If the base object is an array of struct, where each element has pointers, the same mechanism is used for manual deep copy, but it requires a loop to copy and attach each array element pointer:

```
points1* p2;
...
#pragma acc enter data copyin(p2[0:m])
for (k = 0; k < m; ++k) {
  #pragma acc enter data copyin(p2[k].x[0:p2[k].n],\
                                p2[k].y[0:p2[k].n], p2[k].z[0:p2[k].n])
}
#pragma acc parallel loop default(present) ...
for (k = 0; k < m; ++k) {
    for (i = 0; i < p2[k].n; ++i) {
        p2[k].x[i] = ....
    }
}
```

Similar code in reverse will detach the pointers when they appear in a matching `exit data` directive.

What is Missing

There are three things missing with the OpenACC 2.6 manual deep copy definition for C++. These will probably be prototyped in the PGI compilers over the next year or two, and eventually be fixed in a subsequent version of OpenACC.

One is the option to attach one pointer relative to another. Let's look at a simple struct with three pointers, as above, but where one points to the beginning of the allocated space, one to the end, and one somewhere in the middle, perhaps where current processing should start or end.

```
template<typename T>:
typedef class{
    T *start, *end, *current;
}mvect;

mvect v1, v2;
v1.start = new float[n];
v1.end = v1.start + n;
...

#pragma acc enter data copyin(v1)
#pragma acc enter data copyin(v1.start[0:v1.end-v1.start])
for (v1.current = v1.start; v1.current < v1.end; ++v1.current) {
    #pragma acc enter data attach(v1.current)
    #pragma acc parallel loop ...
    for (i = ...) {
        v1.current[i] = ...
    }
}
```

Here, the `enter data attach` directive in the loop will attach and reattach the `v1.current` pointer on the device to its new value every time through the loop. It will initially be attached to the device copy of `v1.start[0]`, then `v1.start[1]`, and so on until `v1.start[n-1]`. In fact, the value of `v1.current` is always an offset from `v1.start` in the vector based at `v1.start`. Here, the `v1.current` pointer always points to a value that is present on the device. However, the `v1.end` pointer actually points to a value past the end of the v1.start block of data. Therefore, v1.end is not present on the device. Even worse, we might allocate v2 right after v1, and get the v2 data block adjacent to and just after the v1 data block. In that case, on the host, v1.end may be equal to v2.start. If we move the two structs to the device, what we want is `v1.start` and `v2.start` to point to the starting position of the two blocks, which are likely to not be adjacent, and `v1.end` and `v2.end` to point just past the v1.start and v2.start blocks of data. In other words, we want a way to tell the compiler or runtime to attach `v1.end` and `v2.end` relative to the starting pointers, `v1.start` and `v2.start`, regardless of whether the v1.end and v2.end addresses are themselves present on the device. This is a known missing piece, and as I mentioned, we will likely prototype something in an upcoming PGI release and have it ready for the next OpenACC version.

The second missing piece is support for the standard template library data types, specifically the `std::vector` class. There are two problems there, one of them being the previous missing piece. The common implementations of `std::vector` have three private pointers, a vector start pointer, a vector end pointer, and a storage end pointer, where one or both of the latter two pointers point just past the allocated memory. The other problem is the `std::vector` class is implemented in a header file with private members, so support for `std::vector` means providing a modified header file with appropriate directives.

The third missing piece is support for user-controlled true deep copy. This is something the OpenACC language committee has been discussing for over three years. Right now we are waiting for a prototype implementation, just as we did with manual deep copy, to shake out the problems with the proposed features and directives. Expect more news about this over the next year.

The next section discusses Fortran-specific issues. You can continue to that, or jump to the last section about particular problems and pitfalls that you should avoid.

Deep Copy in OpenACC for Fortran Programmers

OpenACC was initially designed to port computational kernels from a host to an accelerator device, where the kernels operate on arrays or vectors of data. Many modern programs have much more complex data structures, such as a derived type which contains allocatable or pointers members. Some of these other members may themselves be derived types or arrays of derived type with more allocatable or pointer members.

One solution for this problem with NVIDIA Tesla GPUs has been to use managed memory, enabled in the PGI compilers with the `-ta=tesla:managed` compiler flag on the compile and link steps. With this flag, the allocate statement will put all allocatable or pointer data in managed memory (CUDA Unified Memory), which gets paged on demand between system and device memory by the NVIDIA GPU device driver, and the addresses are the same in the two memories. In many cases, this is a sufficient solution. However, there can be downsides to using managed memory everywhere. First, the cost of a managed memory allocate operation is more expensive than a simple host allocate. Also, data is moved between the memories on demand; there's no support for prefetching data ahead of time, or overlapping data movement with computation. If managed memory is not sufficient, then you need to be able to explicitly control data movement between the memories. With dynamic data structures, you need the pointers in device memory to point to the data in device memory.

Until OpenACC 2.6, there was no standard way to copy such a dynamic data structure to device memory and fix the pointers in device memory. PGI had implemented a first step to solve this problem in the 2017 releases, but it wasn't part of the specification and there were some missing features. As PGI 17.10 and subsequent releases implement more of OpenACC 2.6, we have made important steps to support dynamic data structures.

Attaching Allocatables and Pointer members on the Device

Let's start with a first simple example. Suppose I have a derived type with two allocatable and one pointer array:

```
type points1
  real, allocatable, target :: x(:), y(:)
  real, pointer :: z(:)
  real :: coefx, coefy, coefz
end type
```

Now suppose I've initialized this derived type on the host, and now I want to copy this to and process this on the device. With OpenACC 2.5, the solution was to flatten the derived type:

```
type(points1) :: p1
...
real, pointer :: px(:), py(:), pz(:)
px => p1%x
py => p1%y
pz => p1%z
!$acc enter data copyin(px, py, pz)
!$acc parallel loop default(present) ...
do i = 1, ubound(px,1)
    px(i) = ...
enddo
```

While this works, it requires rewriting the loops that refer to `p1%x(i)` so that they instead use `px(i)`, and so on. It also doesn't scale, if we change the member arrays `x`, `y`, or `z` so they are derived types that contain other pointers.

With OpenACC 2.6, we can write this to refer to the derived type members directly on the device, as long as the hidden data pointers and descriptors in device memory get replaced to point to the device data. We call this the attach operation, to set the hidden pointer for an allocatable or pointer variable or array on the device to point to the corresponding device data for the host pointer. There are several ways to do this. One way is to copy each of the members and the base derived type, then to manually attach the pointers using the `acc_attach` API routine:

```
type(points1) :: p1
...

!$acc enter data copyin(p1%x, p1%y, p1%z)
!$acc enter data copyin(p1)
call acc_attach(p1%x);
call acc_attach(p1%y);
call acc_attach(p1%z);
!$acc parallel loop default(present) ...
do i = 1, ubound(p1%x,1)
    p1%x(i) = ...
enddo
```

The first `enter data` directive copies each of the data arrays to the device. The `second enter data` directive copies the derived type; this copies the derived type directly from system memory, so the `p1%x`, `p1%y`, and `p1%z` pointers still contain host addresses. The `acc_attach` API routine is passed the the pointer to be attached. If the pointer is present on the device as well as the pointer target, the routine replaces the host pointer with the corresponding device pointer in device memory.

A second way to do this is to use the new OpenACC 2.6 attach clause; I emphasize that this clause will be available in the PGI compilers early in 2018:

```
type(points1) :: p1
...
!$acc enter data copyin(p1%x, p1%y, p1%z)
!$acc enter data copyin(p1) attach(p1%x, p1%y, p1%z)
!$acc parallel loop default(present) ...
do i = 1, ubound(p1%x,1)
    p1%x(i) = ...
enddo
```

A third way, and simpler to do this uses the new OpenACC 2.6 implicit attach behavior; this is available in the PGI 17.10 compilers:

```
type(points1) :: p1
...
!$acc enter data copyin(p1)
!$acc enter data copyin(p1%x, p1%y, p1%z)
!$acc parallel loop default(present) ...
do i = 1, ubound(p1%x,1)
    p1%x(i) = ...
enddo
```

The difference between this version and the previous is the base structure is copied to device memory first. When the member pointers are copied in the second enter data directive, the implementation will test if the pointers themselves (`p1%x`, `p1%y`, `p1%z`) are also present. In this case, they are present because of the first `enter data` directive. Because the pointers are present, they are implicitly attached to the `x`, `y`, and `z` device data.

Detaching Pointers on the Device

When done with the processing on the device, the data may need to be copied back to system memory, or at least deallocated. This is done with matching `exit data` directives with either `copyout` or `delete` directives. One potential problem is to make sure the pointers in the device copy of the base structure are detached (that is, restored to the original host pointer values) before the base derived type gets copied back to the host. As with the attach operation, there are three ways to do the detach operation. The first is to manually detach the pointers using the `acc_detach` API routine:

```
call acc_detach(p1%x);
call acc_detach(p1%y);
call acc_detach(p1%z);
!$acc exit data copyout(p1%x,p1%y,p1%z)
!$acc exit data copyout(p1)
```

The `acc_detach` API routine has the same interface as acc_attach. If that hidden pointer is present on the device and attached to a device address, it will be detached and restored to the host pointer value. Of course, if the base derived type, `p1` here, is simply deleted, the detach operation isn't necessary.

A second way to do the same operations is to use the new OpenACC 2.6 detach clause:

```
    !$acc enter data detach(p1%x, p1%y, p1%z)
    !$acc enter data copyout(p1%x, p1%y, p1%z)
    !$acc enter data copyout(p1)
```

A third, simpler way uses the implicit detach behavior, again available in the 17.10 release:

```
    !$acc enter data copyout(p1%x, p1%y, p1%z)
    !$acc enter data copyout(p1)
```

It is important that the members are processed before the base derived type, so the hidden pointers in the base derived type can be detached first.

Array of Derived Type

If the base object is an array of derived type, where each element has allocatable or pointer members, the same mechanism is used for manual deep copy, but it requires a loop to copy and attach each array element pointer:

```
    type(points1), allocatable :: p2(:)
    ...
    !$acc enter data copyin(p2(0:m))
    do k = 1, m
      !$acc enter data copyin(p2(k)%x, p2(k)%y, p2(k)%z)
    enddo
    !$acc parallel loop default(present) ...
    for (k = 0; k < m; ++k) {
        for (i = 0; i < ubound(p2(k)%x); ++i) {
            p2(k)%x(i) = ....
        }
    }
```

Similar code in reverse will detach the pointers when they appear in a matching `exit data` directive.

What is Missing

There is still one thing missing with the OpenACC 2.6 manual deep copy definition for Fortran, support for true user-controlled deep copy. This is something the OpenACC language committee has been discussing for over three years. Right now we are waiting for a prototype implementation, just as we did with manual deep copy, to shake out the problems with the proposed features and directives. Expect more news about this over the next year. Right now, the PGI compilers have a command line flag to enable implicit full deep copy, where all allocatable or pointer members of a derived type will be processed whenever a derived type is processed. The user control will let a programmer select which members to copy, and to control which direction to copy which members. We hope to prototype this in the PGI compilers over the next year, and eventually include this in a subsequent version of OpenACC.

Pitfalls

When using manual deep copy, whether for C++/C or Fortran, there are a few details that you might need to be aware of. Most of the time, the manual deep copy will work as expected.

Be Careful With Updates

It is important to note that while the base struct or derived type is on the device with the pointers attached, that an update directive to update the host values of the struct or derived type itself from the device will copy those device pointers to the host copy, which can produce invalid memory reference on the host. An `update` directive to update the device copy of the struct or derived type from the host will copy the host pointers to the device copy of the struct, which again will result in invalid memory references on the device when those pointers are used.

Attach Counter

The `acc_attach` API routines or the attach clause or the implicit attach operation on other data clauses actually keep track of how many attach operations have been processed for that pointer. The problem this solves is illustrated by the following example:

```
typedef class{
    float *x, *y, *z;
    float coefx, coefy, coefz;
    size_t n;
}points1;

points1 p1;
...
#pragma acc enter data copyin(p1)
#pragma acc enter data copyin(p1.x[0:p1.n], p1.y[0:p1.n], p1.z[0:p1.n])
...
foo(&p1);
...
#pragma acc exit data copyout(p1.x[0:p1.n], p1.y[0:p1.n], p1.z[0:p1.n])
#pragma acc exit data copyout(p1)

void foo(points1* q){
    #pragma acc enter data copyin(q[0:1])
    #pragma acc enter data copyin(q->x[0:q->n], \
                                  q->y[0:q->n], q->z[0:q->n])
    ...
    #pragma acc exit data copyout(q->x[0:q->n], \
                                  q->y[0:q->n], q->z[0:q->n])
    #pragma acc exit data delete(q[0:1])
}
```

In the routine foo, the struct for `q` is already present, and the pointers were already attached. We don't want to attach them again, because the attach operation requires a data transfer from host to device, and we'd like to avoid that when possible. By keeping track of whether the pointers are attached, we can avoid the transfer for the second enter data. The first enter data for the pointers `p1.x`, `p1.y`, and `p1.z` will increment the attach counters from 0 to 1. In routine foo, because the pointers are already attached, no transfer will occur and the attach counters will be incremented from 1 to 2.

At the end of that routine, we don't want to detach the pointers, because they could be used after the call to foo by the caller. Because the attach counters are greater than one, they will only be decremented here. The exit data operation in the caller will decrement the attach counters to zero, which then causes the pointers to be detached.

Finalize

The discussion just above discusses what happens at an `exit data` directive when the attach counter for a pointer is greater than one. If the `exit data` directive has the finalize clause and has a reference to a struct or class member pointer, or derived type allocatable or pointer member, the attach count is immediately set to zero and the pointer will be immediately detached. There is a corresponding runtime API routine, `acc_detach_finalize`, to perform the same operation. This may be useful when you need a pointer immediately detached.

Summary

OpenACC 2.6 has new support for manual deep copy of complex dynamic data structures. PGI 17.10 has support for most of these features, and the rest will be coming soon. The new attach and detach behavior for class, struct, or derived type members is supported for the data clauses on any directive or construct. Keep track of PGI announcements for other features that support dynamic data structures in OpenACC.