Transferring user-defined types in OpenACC

James Beyer, David Oehmke, Jeff Sandoval
Cray, Inc.
Saint Paul, USA
(beyerj|doehmke|sandoval)@cray.com

***Abstract*—A preeminent problem blocking the adoption of OpenACC by many programmers is support for user-defined types: classes and structures in C/C++ and derived types in Fortran. This problem is particularly challenging for data structures that involve pointer indirection, since transferring these data structures between the disjoint host and accelerator memories found on most modern accelerators requires deep-copy semantics. This paper will look at the mechanisms available in OpenACC 2.0 to allow the programmer to design transfer routines for OpenACC programs. Once these mechanisms have been explored, a new directive-based solution will be presented. Code examples will be used to compare the current state-of-the-art and the new proposed solution.**

***Keywords*-OpenACC; user-defined types; accelerators; compiler directives; heterogeneous programming**

I. Introduction

A preeminent problem blocking the adoption of OpenACC by many programmers is support for user-defined types: classes and structures in C/C++ and derived types in Fortran. User-defined types are an important part of many user applications; for example, in C and C++ something as simple as a complex number is really a structure of two floats. This problem is particularly challenging for data structures that involve pointer indirection, since transferring these data structures between the disjoint host and accelerator memories found on most modern accelerators requires deep-copy semantics. Unfortunately, the current OpenACC specification only allows for shaping the top-level of an array; there is no mechanism for shaping a pointer inside of a structure. This problem is manageable for Fortran, where most objects are self-describing, but unavoidable for C and C++, where all pointers are unshaped.

There are really only two solutions to this problem available in OpenACC 2.0 [1]: (1) refactor the code to stop using pointers in structures (i.e. use arrays of structures to replace structures of arrays) and (2) use an API to move the objects to the device.

The second solution is preferable because it affects only code that performs data transfers rather than all code that makes use of the user-defined type. This paper will look at the mechanisms available in OpenACC 2.0 to allow the programmer to design transfer routines for OpenACC programs. Using a simple example we will show that this approach requires writing large amounts of low-level code to manage device memory, which defeats the goal of a high-level directive-based programming model. In fact, the resulting code is very similar to that required when writing a CUDA program [2]. For OpenACC, these details should instead be handled by the compiler.

The Cray compiler provides a deep-copy command line option for Fortran [3]. However, experience has shown that users often have extremely large data sets contained in their derived types and only want to transfer the parts that they need for a given kernel. As a result, this paper will present a more flexible directive-based solution. Code examples will be used to compare the current state-of-the-art and the new proposed solution. The directives we propose provide a mechanism to shape an array contained inside of a structure, which will bring C and C++ programs to the same level as Fortran concerning deep copies. This same mechanism can also be used to limit which parts of a user defined object are transferred, avoiding the all-or-nothing problem encountered with the Cray deep- copy option. Along with elegantly solving the shaping problem, this mechanism also reduces the level of difficulty for the programmer since they only need to express the shape of a pointer rather than program how to move the memory behind the pointer.

II. Background and problem

OpenACC has limited support for user-defined types: a variable of user-defined type must be a *flat* object – that is, it may not contain pointer indirection (including Fortran allocatable members). This restriction simplifies implementations, ensuring that all variables are identical on both host and device and may be transferred as contiguous blocks of memory. However, feedback from the OpenACC user community indicates that this restriction is a significant impediment to porting interesting data structures and algorithms to OpenACC. This section describes the challenges to relaxing this restriction.

Variables and data structures that contains pure data are interpreted the same on the host and device – therefore, an implementation may transfer such a variable to and from device memory as a simple memory transfer, without regard for the contents of that block of memory.[1] On

[1] Although it is technically possible for a host and device to interpret pure data differently, architectures that follow common conventions and standards (e.g., equivalent endianness, two's complement and IEEE floating point) are compatible.

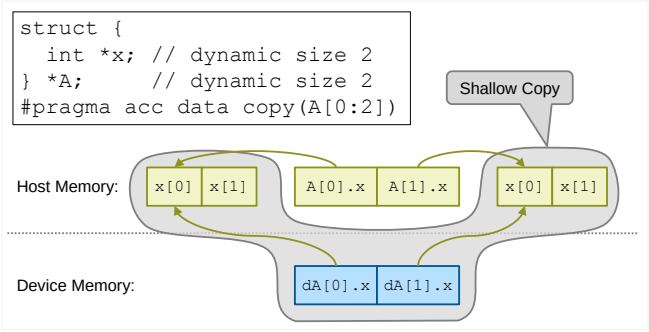


Figure 1. Shallow copy

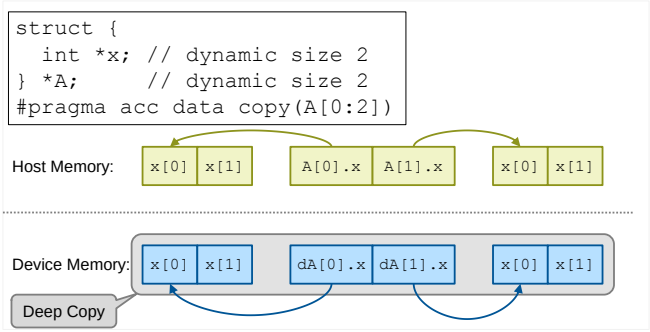


Figure 2. Deep copy

the other hand, data structures that contain pointers are interpreted differently on the host and device. A host pointer targets a location in host memory and a device pointer targets a location in device memory; either pointer may be legally dereferenced only on its corresponding device.[2] If an implementation transfers a data structure using a simple transfer, then pointers in that data structure will refer to invalid locations – this transfer policy is referred to as *shallow* copy, which is illustrated in Figure 1. For comparison, shallow copy is acceptable for shared-memory programming models like OpenMP because it is always legal to dereference a pointer (although shallow-copy does imply that the target of member pointers will be shared among all copies). In contrast, shallow copy is less useful in OpenACC because dereferencing the resulting pointer may not be legal. Instead, users often require deep-copy semantics, as illustrated in Figure 2, where every object in a data structure is transferred. Deep copy requires recursively traversing pointer members in a data structure, transferring all disjoint memory locations, and translating the pointers to refer to the appropriate device location. This technique is also known as object serialization or marshalling, which is commonly used for sending complex data structures across networks or writing them to disk. For example, Cray's first deep-copy prototype implementation was inspired by MPI, which has basic support for describing the layout of user-defined data types and sending user-defined objects between ranks [4].

A Fortran compiler could automatically perform deep copy, since Fortran pointers are self-describing dope vectors. In fact, deep copy is the language-specified behavior for intrinsic assignment to a derived type containing allocatable members. Unfortunately, a C/C++ compiler cannot automatically perform deep copy, since C/C++ pointers are raw pointers that do not contain shape information. Further, even for self-describing dope-vectors in Fortran, a user may desire to copy only some subset of deep members. Addressing these problems requires additional information from the programmer.

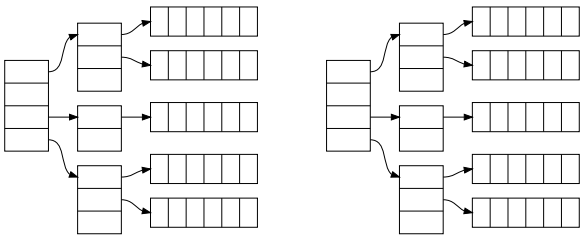
Finally, throughout the paper we will distinguish between two types of members for user-defined types: *direct members* and *indirect members*. Direct members store their contents directly in the contiguous memory of the user-defined type. A scalar or statically-sized array member is a direct member. In contrast, *indirect members* store their contents outside of the contiguous memory of the user-defined type, where it is accessed through pointer indirection. Thus, an indirect member always has a corresponding direct member that stores the address or access mechanism of the indirect part. For example, a pointer member in C/C++ is a direct member, but the target of the pointer is an indirect member. In Fortran, an allocatable member is an indirect member, but the underlying dope vector for the allocatable data is a direct member. Since direct members are contained directly in an object, they are automatically transferred as part of that object. But indirect members are not contained directly in an object, and thus require deep-copy semantics to correctly transfer them.

A. Desired capabilities

There are several distinct capabilities that we can attempt to offer users to improve support for user-defined types with indirect members. These techniques are all various forms of deep-copy, where every sub-object on the device must be accessible through a chain of parent objects originating at a single base object – this ensures that addressing calculations and access mechanisms are identical for both host and device code. Additionally, these techniques all assume that individual objects are transferred in whole and have an identical layout in both host and device memory.

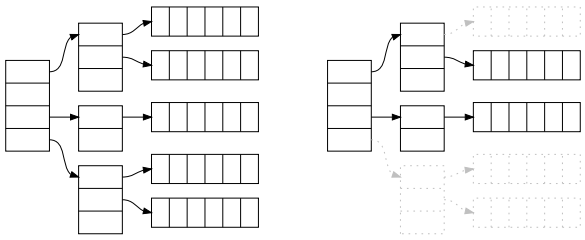
Member shapes: Allowing users to explicitly shape member pointers puts C/C++ on the same footing as Fortran: pointers become *self describing*. This first step is important because it makes automatic deep-copy possible. However, member shapes aren't useful alone, since they don't imply allocation or transfer – instead, they must be used with other

[2]Pointers are functionally equivalent if a host and device share common memory. Even so, dereferencing such a non-local pointer will likely affect performance due to memory locality; thus, on these devices it may still be desirable to physically transfer memory targeted by pointers.



(a) Host memory (b) Device memory

Figure 3. Full deep copy



(a) Host memory (b) Device memory

Figure 4. Selective member deep copy

data clauses to achieve deep-copy semantics.

In most cases, member shapes should be global properties of a user-defined type, accessible to all data regions that transfer objects of that type. However, member shapes could certainly be defined as a local property, only affecting data regions at the same program scope.

Full deep-copy: With self-describing dope-vectors in Fortran and user-supplied shape declarations in C/C++, an implementation can automatically perform deep copy for user-defined types with indirect members. This approach is known as *full deep-copy*, since it results in a complete duplication of a host data structure in device memory. Figure 3 shows a full deep copy from (a) host memory to (b) device memory – the entire data structure is replicated.

In contrast to shallow copy, which only requires replicating an individual object, deep copy requires replicating all sub-objects in a data structure that are accessible from a common base object. Deep copy also requires replicating all object-to-object pointer relationships, which is done by initializing pointer members in the replicated objects.

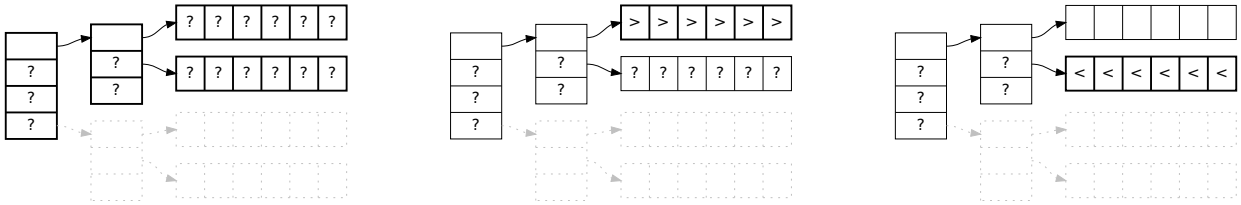
Although we describe deep-copy in terms of `copy`-clause behavior, the concept is applicable to all of the data clauses. Deep `create` replicates all objects in a data structure, but the data members need not be initialized. However, unlike shallow `create`, where no transfer is required, deep `create` requires transfers to properly initialize member pointers. Deep `free` deallocates all objects in a data structure and deep `update` updates all objects in a data structure. Finally, deep `copyin` and deep `copyout` can be described as combinations of deep `create`, deep `update` and deep `free`.

The `present` clause could imply either shallow or deep semantics. Shallow `present` implies a present check only for the top-level object in a data structure – it is programmer responsibility to ensure that all required sub-objects are also present and properly accessible through member pointers. On the other hand, deep `present` implies a present check for all objects in a data structure. However, presence alone does not indicate that a data structure is available for use – object-to-object member pointers must also be properly configured. Ensuring that these pointer relationships are valid could be handled by the programmer or the implementation. In either case, member pointers must be properly initialized for the data structure to be usable as expected.

Full deep-copy is most appropriate for user-defined types that represent indivisible collections of members, commonly seen in pure object-oriented programming styles. Such an object is only useful when all or most of its members are available. In this case, full deep-copy can create a complete clone of the object, rendering that object fully usable on the device. As an example, support for member shapes and full deep-copy would make it possible to provide OpenACC-aware versions of the common C++ STL containers.

Selective members: Full deep-copy is not appropriate for user-defined types that are used for coarse-grained organizational purposes. These objects are rarely treated as single entities, but instead various subsets of the members are used in different contexts. In this case, it is very inefficient to make a complete clone of the object, since many or most of the members are not needed at one time. Instead, it is preferable to allow users to select different members to include and exclude in different data regions, a capability that we refer to as *selective member* deep-copy. This capability requires a mechanism for specifying different deep-copy policies based on context.

Selective member deep-copy differs from full deep-copy in that some pointers in a host data structure will not be traversed during a transfer, instead leaving invalid pointers in the device data structure. Of course, a user requesting selective deep-copy is responsible for only accessing members that are transferred – accessing a member that is not transferred is considered a programmer error, just like accessing an out-of-bounds value. Figure 4 shows selective member deep-copy, where the greyed-out objects are not transferred to device memory. Selective member deep-copy saves device memory and transfer bandwidth by skipping some sub-objects that are not referenced on the device. However, every present sub-object must be accessible through a chain of objects originating at a single base object – this ensures that addressing calculations and access mechanisms



(a) Deep `create` (b) Selective `copyin` (c) Selective `copyout`

Figure 5. Selective direction deep copy

are identical for both host and device code.

Selective member deep-copy in an inherently local operation, since different contexts in a program may require different deep-copy behaviors. This differentiation can be achieved in various ways, however. One possible solution is to use local member shapes to shape member pointers differently in different contexts. This requires a means of canceling or overriding any global member shapes that may be visible. Another possible solution is to allow global *named policies* to be declared for a particular user-defined type. Then different contexts may request policies for the user-defined type. Ultimately, the end result is the same for both solutions – a data structure is partially replicated in device memory.

Selective directionality: For scalars and arrays of primitive type, a single data-clause behavior (e.g., `create`, `copyin`, `copyout`, etc.) is usually sufficient for an entire object. However, because user-defined types are a collection of individual members, it is reasonable to expect that different members will require different data-clause behavior. For example, consider an object where one member is read-only and another is write-only. Transferring this entire object with a `copy` clause is functionally sufficient, but using a `copyin` clause for the read-only member and a `copyout` clause the write-only member could significantly reduce transfer bandwidth. As a result, it may be useful to provide mechanisms for specifying different members in different data clauses. We refer to this as *selective direction* deep-copy, since different members will transfer in different directions.

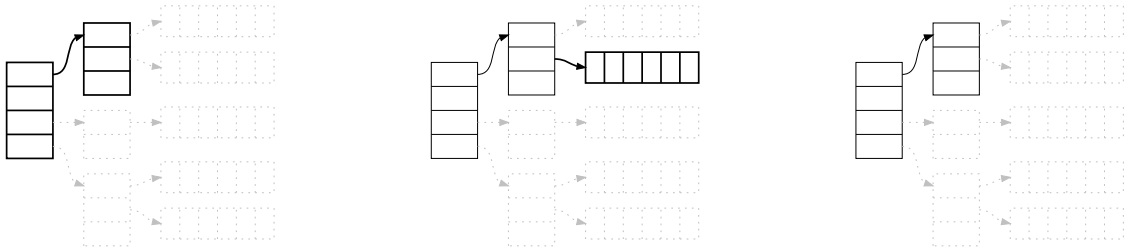
Conceptually, selective direction deep-copy is a multi-step process[3], as illustrated in Figure 5. First, a data structure is allocated on the device with deep-`create` semantics, only initializing the required pointers and leaving all other elements uninitialized. This step is shown in Figure 5a, with "?" labels indicating uninitialized data. Next, the `copyin` sub-objects are transferred to device memory, shown in Figure 5b with $>$ labels. Then, at the end of the data region the `copyout` sub-objects are transferred back to host memory, shown Figure 5c with $<$ labels. Only the parts of the data structure that are used on the device need to be initialized or transferred – other than for active pointer members, the two parent objects remain uninitialized throughout the data region.

Note that selective directionality is not required for functionality. A user can always use the data-clause behavior that satisfies all references to all members and sub-objects. For example, using a `copy` clause is sufficient for a user-defined object that has one read-only member and another write-only member. However, this approach is coarse grained and may issue unnecessary transfers in both directions.

Moreover, note that selective directionality can always be achieved through `update` directives. Accomplishing this requires two steps: (1) issue a deep `create` to allocate an uninitialized clone of a data structure and (2) issue individual `update` directives to move specific members in the desired direction. When the data structure is no longer needed it will be deep deallocated. The only caveat with using `update` directives is that they are unconditional, so using them with a `present_or_create` clause can be awkward.

Mutable deep-copy: Even with selective member and direction, deep copy treats a user-defined object as a single entity, particularly with respect to allocation and deallocation. That is, a base object and all required sub-objects are allocated and deallocated at the same time – the composition of a data structure is immutable throughout its lifetime. However, there may be times when a user does not want to assemble or disassemble a data structure in a single step; instead, they may want to assemble or disassemble it over a period of time, using a series of distinct steps. For example, a data structure may be so large that only a subset of its members fit in device memory at one time. A user may want to copy one group of sub-objects for one part of a data region and another group of sub-objects for another part of the same data region, all while leaving a third group of sub-objects on the device for the entire data region. Because it allows modifying the composition of a data structure during its lifetime, we call this capability *mutable deep-copy*.

[3]In practice, the steps at the start the data region can be merged into a single operation, as can the steps at the end of the data region.



(a) Selective member deep-copy (b) Attach to parent (c) Detach from parent

Figure 6. Top-down mutable deep-copy

Mutable deep-copy allows a user to *attach* objects to and *detach* them from one another, independently of allocating or transferring those objects. That is, an object may attach to an existing parent object or an existing sub-object. This capability differs from other deep-copy capabilities, which require the composition of a data structure remain fixed throughout a data region.

Pointer translation for mutable deep-copy can be either top-down or bottom-up. Top-down translation occurs when an object is attached to its existing parent object – this paradigm allows a user to assemble a data structure in a top-down manner. Figure 6 illustrates a three-step top-down mutable deep-copy. First, selective deep-copy transfers a base object and a sub-object; then, another sub-object is attached to that data structure; finally, that sub-object is detached from the data structure, leaving only the original base object and single sub-object. In contrast, bottom-up translation occurs when an object is attached to its existing sub-objects – this paradigm allows a user to assemble a data structure in a bottom-up manner. Figure 7 illustrates bottom-up mutable deep-copy. First, a set of sub-objects are transferred to the device, independent of their parent objects. Next, another set of sub-objects are transferred to the device, some of which attach to other existing sub-objects. Finally, the top-level base object is transferred to the device and attaches to existing sub-objects.

B. Data layout

Programmers often misunderstand the distinction between direct and indirect members. For example, they don't always realize that direct members are automatically included when transferring an object of user-defined type. Instead, they sometimes want to selectively transfer a subset of direct members of a user-defined type, a capability that at first glance appears very similar to selective-member deep-copy. But beneath the surface they are vastly different capabilities.

Because a direct member is contained directly in the contiguous memory of a user-defined type, it is always accessed at a fixed offset from the start of an object. If direct members were allowed to be placed on the device selectively, then any missing direct members would skew the offsets for subsequent members. We refer to this capability as *data layout*.

Changing the data layout of a variable essentially changes the type of that variable, since it removes some members and affects the offsets and addressing calculations for the remaining members. Although not impossible, supporting data layout for anything more than simple cases is difficult and beyond the scope of this paper.

On the other hand, deep-copy applies to indirect members. Every object and sub-object involved in a deep-copy operation is transferred in whole, including all direct members. This preserves equivalent offsets and address calculations between host and device. Selective-member deep-copy simply makes some indirect members unavailable, but the corresponding direct member (i.e., the pointer) is still available, even if it may not be legally dereferenced.

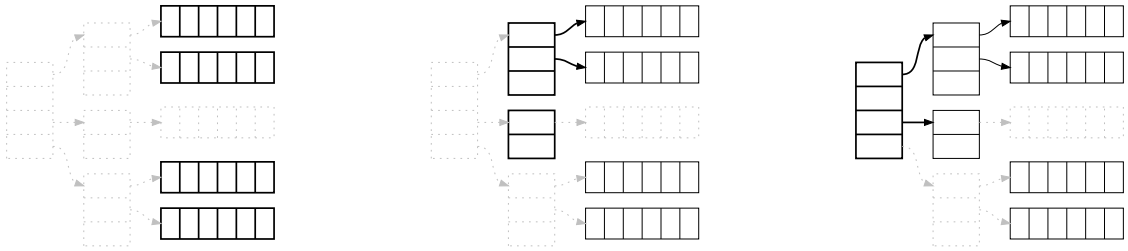
The primary motivation for data layout is for large user-defined types, containing many large direct members. If only a small fraction of those members are needed on the device, then programmers would prefer to save the memory and bandwidth that allocating and transferring them would consume. Although the memory cannot be saved without data layout, the transfer bandwidth can be. For example, an object can always be allocated in full but selectively transferred, allowing the programmer to reduce transfer bandwidth by skipping members that are not needed.

C. Motivation

After the completion of OpenACC 2.0, the dominant user request heard by the OpenACC language committee has been support for deep copy. This section describes one real-world example where deep-copy is required for porting an application to OpenACC.

ICON, a climate code developed by the German Weather Service (DWD) and the Max Planck Institute for Meteorology (MPI-M), uses derived types that must be made available on the device.

For example, one derived type is `t_nh_state`, declared in Figure 8. For CCE, the shallow size of this derived type is



(a) Copy sub-objects (b) Attach to existing children (c) Attach to existing children

Figure 7. Bottom-up mutable deep-copy

```
TYPE t_nh_state

  !array of prognostic states at different timelevels
  TYPE(t_nh_prog),  ALLOCATABLE :: prog(:)       !< shape:
       (timelevels)
  TYPE(t_var_list), ALLOCATABLE :: prog_list(:)  !< shape:
       (timelevels)

  TYPE(t_nh_diag)     :: diag
  TYPE(t_var_list)    :: diag_list

  TYPE(t_nh_ref)      :: ref
  TYPE(t_var_list)    :: ref_list

  TYPE(t_nh_metrics) :: metrics
  TYPE(t_var_list)    :: metrics_list

  TYPE(t_var_list), ALLOCATABLE :: tracer_list(:) !< shape
       : (timelevels)

END TYPE t_nh_state
```

Figure 8. ICON data structure

roughly 19KB, but the deep size of this derived type is much larger because the member `metrics` contains roughly 84 pointer/allocatable members and the member `diag` contains roughly 86 pointer/allocatable members. Moreover, only the following members are needed on the device at one time:

```
p_nh_state(:)%metrics%rayleigh_w(:)
p_nh_state(:)%metrics%rayleigh_vn(:)
p_nh_state(:)%diag%vn_ie(:,:,:)
p_nh_state(:)%diag%vt(:,:,:)
p_nh_state(:)%diag%dvn_ie_ubc(:,:)
p_nh_state(:)%diag%e_kinh(:,:,:)
p_nh_state(:)%diag%w_concorr_c(:,:,:)
p_nh_state(:)%prog(:)%vn(:,:,:)
```

Because a relatively small number of pointer/allocatable members are actually referenced on the device, this data structure is a poor candidate for full deep copy – OpenACC developers of this code agree with this assessment. However, selective deep-copy may be a viable option. All of the members referenced represent a single level of indirection except for `p_nh_state(:)%prog(:)%vn(:,:,:)`, which represents a selective deep copy of derived type `type(t_nh_prog)`. This size of this type is roughly 1KB per element, plus the additional selective deep copy of member `vn`. Selective directionality could be helpful for this code, since some members are read-only and others are read-write.

Additionally, this data structure contains many *convenience pointers*, which are designed to alias other members in the data structure. This aliasing introduces an ordering complexity, since the aliases must be processed after the members to which the aliases point. Section IV-A1 describes a solution to this problem.

Finally, the ICON data structure contains embedded linked lists. Fortunately, the developers indicate that these linked lists are not currently needed on the device, since they are only used for I/O. Even so, it is likely that other applications will require transferring recursive data structures to an accelerator, so it is prudent to keep this use-case in mind when designing deep-copy support.

III. Existing Solutions

A. Refactoring

The most basic way a programmer can avoid deep-copy related problems with OpenACC is to rewrite their application to not require it. For example, user-defined types with pointer members would need to be converted to equivalent user-defined types with statically-sized arrays. This approach of restructuring and packing a data structure is often used by CUDA programmers to reduce transfer costs for data structures used on the device.

However, porting an application to CUDA requires rewriting it. Given such a large effort, the additional incremental effort to rewrite the data structures at the same time is small. In contrast, porting an application to OpenACC only requires adding directives. In this case, the additional incremental effort to rewrite the data structures is much larger. Thus, refactoring an application to avoid deep-copy is an undesirable solution.

B. API

OpenACC 2.0 introduced a function API that provides low-level data transfer functions [1]. A programmer can use these functions to perform deep copy manually [5].

```
struct Deep {
 int size;
 double scalar;
 double* A; /* A[0:size] */
 double* B; /* B[0:size] */
};

void deep_copy(struct Deep* P, int n) {
 int i,j;
 struct Deep* dP;
 double* dA;
 double* dB;

 /* enter copyin(P[0:n]) */
 dP = acc_copyin(P, sizeof(struct Deep)*n);
 for (i=0; i<n; ++i) {
  /* copyin P[i].A[0:P[i].size] */
  dA = acc_copyin(P[i].A, P[i].size*sizeof(double));
  /* update dP[i].A to point to device copy */
  acc_memcpy_to_device(&dP[i].A, &dA, sizeof(double*));
  /* copyin P[i].B[0:P[i].size] */
  dB = acc_copyin(P[i].B, P[i].size*sizeof(double));
  /* update dP[i].B to point to device copy */
  acc_memcpy_to_device(&dP[i].B, &dB, sizeof(double*));
 }

 /* P is available for use on device */

 /* exit copyout(P[0:n]) */
 for (i=0; i<n; ++i) {
  /* update P[i].scalar (if necessary) */
  acc_update_self(&P[i].scalar, sizeof(double));
  /* copyout P[i].A[0:P[i].size] */
  acc_copyout(P[i].A, P[i].size*sizeof(double));
  /* copyout P[i].B[0:P[i].size] */
  acc_copyout(P[i].B, P[i].size*sizeof(double));
 }
 /* delete P[0:n] */
 /*
  * We can't copyout because pointers A and B have been
  * changed to point to device memory in the device copy.
  * If we copy the entire structure back to the host we
  * will overwrite the host pointers.
  */
 acc_delete(P, sizeof(struct Deep)*n);
}
```

Figure 9. Full manual deep-copy

Figure 9 shows how to manually achieve full deep-copy for a struct containing two pointer members that are shaped by another integer member. The code follows a top-down strategy, first copying the top-level objects. Then, for each second-level sub-object it copies the sub-object and fixes the device pointer in the device copy (since the member pointers in the top-level objects contain host pointers). The pointer fix-up is just a small transfer to move the device pointer into the appropriate location in the device copy of the data structure.

The `copyout` operation is accomplished in a manner similar to the `copyin` operation, but the order is reversed to use a bottom-up strategy. The second-level sub-objects are transferred back to the host and deleted. Then, the top-level objects are transferred back to the host, taking care to avoid overwriting a host pointer with a device pointer – the non-pointer members must be transferred, while the pointer members must be skipped (or reverse-translated from device addresses to host addresses). In general, implementing any of the other data clauses or update directives follows the same pattern as described for this example, but requires writing slightly different code for each operation.

```
struct Deep {
 int size;
 double scalar;
 double* A; /* A[0:size] */
 double* B; /* B[0:size] */
};

void selective_direction(struct Deep* P, int n) {
 int i,j;
 struct Deep* dP;
 double* dA;
 double* dB;

 /* enter copyin(P[0:n], P[0:n].B[]) create(P[0:n].A[]) */
 dP = acc_copyin(P, sizeof(struct Deep)*n);
 for (i=0; i<n; ++i) {
  /* create P[i].A[0:P[i].size] */
  dA = acc_create(P[i].A, P[i].size*sizeof(double));
  /* update dP[i].A to point to device copy */
  acc_memcpy_to_device(&dP[i].A, &dA, sizeof(double*));
  /* copyin P[i].B[0:P[i].size] */
  dB = acc_copyin(P[i].B, P[i].size*sizeof(double));
  /* update dP[i].B to point to device copy */
  acc_memcpy_to_device(&dP[i].B, &dB, sizeof(double*));
 }

 /* P is available for use on device */
 /* P[:].A[:] = P[:].B[:]*P[:].scalar */

 /* exit delete(P[0:n], P[0:n].B[]) copyout(P[0:n].A[]) */
 for (i=0; i<n; ++i) {
  /* copyout P[i].A[0:P[i].size] */
  acc_copyout(P[i].A, P[i].size*sizeof(double));
  /* delete P[i].B[0:P[i].size] */
  acc_delete(P[i].B, P[i].size*sizeof(double));
 }
 /* delete P[0:n] */
 /* can't copyout because pointers A and B
    are changed on device */
 acc_delete(P, sizeof(struct Deep)*n);
}
```

Figure 10. Selective-direction manual deep-copy

Figure 10 shows how to manually achieve selective-direction deep-copy for the same struct used in Figure 9. Just like that example, the code employs a top-down strategy at the start of the logical data region and a bottom-up strategy at the end of it. First, it performs a `copyin` operation on the top-level objects. Then, it either performs a `create` or `copyin` operation on the second-level objects, depending on how those objects are used on the device. Since member `A` is computed on the device it doesn't need to be transferred to device memory. But, member `B` is read-only on the device, so it must be allocated and transferred. However, in both cases the pointer members in the top-level objects still must be properly initialized.

Again, the `copyout` process is reversed, following a bottom-up strategy. This time member `A` is transferred back to host memory and deleted, while member `B` can just be deleted (since it was not modified). Finally, just like the last example care must be taken to not overwrite the member pointers in the top-level host object with device pointers.

```
struct Deep {
 int size;
 double scalar;
 double* A; /* A[0:size] */
 double* B; /* B[0:size] */
};

void selective_member(struct Deep* P, int n) {
 int i,j;
 struct Deep* dP;
 double* dA;

 /* enter copyin(P[0:n], P[0:n].A[]) */
 dP = acc_copyin(P, sizeof(struct Deep)*n);
 for (i=0; i<n; ++i) {
  /* copyin P[i].A[0:P[i].size] */
  dA = acc_copyin(P[i].A, P[i].size*sizeof(double));
  /* update dP[i].A to point to device copy */
  acc_memcpy_to_device(&dP[i].A, &dA, sizeof(double*));
 }

 mutable_copy(P, n);

 /* exit delete(P[0:n]) copyout(P[0:n].A[]) */
 for (i=0; i<n; ++i) {
  /* copyout P[i].A[0:P[i].size] */
  acc_copyout(P[i].A, P[i].size*sizeof(double));
 }
 /* delete P[0:n] */
 /* can't copyout because pointers A and B are changed on
      device */
 acc_delete(P, sizeof(struct Deep)*n);
}

void mutable_copy(struct Deep* P, int n) {
 int i;
 struct Deep* dP;
 double* dB;

 /* enter present(P[0:n]) copyin(P[0:n].B[]) */
 dP = acc_deviceptr(P);
 for (i=0; i<n; ++i) {
  /* copyin P[i].B[0:P[i].size] */
  dB = acc_copyin(P[i].B, P[i].size*sizeof(double));
  /* update dP[i].B to point to device copy */
  acc_memcpy_to_device(&dP[i].B, &dB, sizeof(double*));
 }

 /* A and B are both now available on the device */

 /* exit copyout(P[0:n].B[]) */
 for (i=0; i<n; ++i) {
  /* copyout P[i].B[0:P[i].size] */
  acc_copyout(P[i].B, P[i].size*sizeof(double));
  /* restore the pointer on the device to the host version
       */
  acc_memcpy_to_device(&dP[i].B, &P[i].B, sizeof(double*))
      ;
 }
}
```

Figure 11. Mutable manual deep-copy

In this case the top-level object may just be deleted, since the non-pointer members did not change. Notice how similar this code looks to that in Figure 9 – the deep-traversal pattern is the same, while the operations at each step differ.

Finally, figure 11 shows how to manually achieve selective-member and mutable deep-copy for the same struct used in the last two examples. This example differs from the last two in that the composition of the device copy of the data structure changes throughout the logical data region. The selective-member `copyin` functionality is achieved by performing deep-copy for member `A`, but shallow-copy for member `B`. These steps are reversed for the `copyout` operation at the end of the logical data region. For both cases, the pointer for member `B` is treated as raw data – the host pointer is transferred to the device without translation, and it is not transferred back because it is not modified.

The mutable deep-copy comes from another logical data region, nested within the outer one. At the start of the inner region, the top-level object and member `A` is available. This data region essentially transfers member `B` and *attaches* it to the top-level object; at the end of the region, it transfers member `B` back to the host, detaches it from the top-level object, and deletes it. It is important for the detachment operation to restore the member pointer `B` back to the original host pointer, to avoid the possibility of transferring a device pointer back to the host (since the outer data region did not translate that pointer).

The reader should note that performing manual deep copy in Fortran is not quite as straightforward as in C/C++, since performing pointer fix-up requires knowledge about an implementation's underlying dope vector. In C/C++, raw pointers are exposed to the programmer and may be explicitly re-assigned to point to different memory locations. As a result, attach and detach operations reduce to simple pointer assignments of pointers in device memory. On the other hand, Fortran encapsulates raw pointers in an opaque, implementation-defined dope-vector that users cannot modify directly. Further, dope-vector re-association is only supported for true *pointer* variables, but not for *allocatable* variables.

In summary, the OpenACC 2.0 API makes manual deep-copy possible for pointer members. But, this solution is inherently ill-suited to a directive-based programming model because it requires the user to write significant amounts of code. That code is tedious and error-prone, involving deep loop nests, requiring careful address calculations, pointer assignments, and non-contiguous transfers. The code does follow a common boilerplate for all cases, but it requires replication and subtle modification for every distinct data type, deep-copy policy, and context in which it is invoked. Moreover, attempting asynchronous manual deep-copy would add an additional dimension of complexity, not to mention that the current API does not provide the necessarily asynchronous functionality. As a result, we view manual deep-copy as an undesirable and temporary workaround for the lack of deep-copy support in OpenACC.

C. Fortran automatic deep-copy

Because pointer indirection in Fortran is encapsulated in self-describing dope vectors, a Fortran compiler could automatically perform deep copy. As an extension to the OpenACC standard, the Cray Fortran compiler in CCE 8.2 provides a deep-copy option that automates all of the work in

transferring derived types with allocatable and pointer members [3]. Simply adding the `-hacc_model=deep_copy` option enables full deep-copy for pointer and allocatable members in Fortran derived types (the default option is `-hacc_model=no_deep_copy`). Although this feature is not portable across vendors, it is available today in a released and supported product.

Automatic deep-copy is very user-friendly, since it doesn't require any code changes or directives; but, it has several drawbacks that limit its usefulness. First, it enables full deep-copy for all derived types in all contexts – a user has no means of requesting selective deep-copy, which can result in wasted device memory and transfer bandwidth. Second, automatic deep-copy does not support aliasing. This aliasing limitation is fine for allocatable members, which by definition cannot alias, but is problematic for pointer members that alias. Third, and most importantly, automatic deep-copy only applies to Fortran – it is not possible for C/C++, since pointer indirection in these languages is accomplished with raw pointers that do not contain shape information. Moving beyond these limitations requires additional information from the programmer.

IV. Directive-based solution

There are several key design choices one faces when considering a directive-based solution for deep copy. Should the deep-copy behavior be specified locally, explicitly repeated at each data region that requires deep copy, or should it be specified once globally, where it applies to every visible data region? Should deep copy be specified for specific variables, or should it be specified for all variables of some type?

Specifying a deep-copy behavior globally allows that behavior to apply consistently throughout a program; but, it becomes difficult to customize the policy for different contexts. Specifying deep-copy behavior locally allows different behavior for every context, but achieving a consistent policy requires duplicating the same deep-copy specification in many places throughout the program.

Similarly, associating a deep-copy behavior with a particular user-defined type allows consistent behavior for all variables of that type, but makes it difficult to customize the policy for some variables. Associating a deep-copy behavior with a particular variable allows better customization, but consistency across variables of the same type now requires duplicating the same deep-copy specification for multiple variables.

We opted to strike a balance in these design decisions. To achieve the most general and expressive syntax, we chose a relatively low-level, local syntax that applies to specific variables. But for convenience we provide a higher-level, global syntax that allows defining reusable policies that apply to user-defined types. The two forms can be used together, with local syntax taking precedence over global syntax. This allows users to define default, global policies that can be overridden and further customized locally. We believe this approach offers the best balance between expressiveness and convenience.

The next two sections describe our proposed local, low-level syntax and our global policy syntax. Although a full formal specification of our proposed syntax is beyond the scope of this paper, we point out key differences with the existing OpenACC 2.0 specification [1]. Our proposed changes appear relatively simple, but as our examples will show they are quite powerful.

A. Low-level, local syntax

The syntax described in this section is the foundation for our deep-copy solution. It is meant to be general enough to express all of the capabilities described in Section II-A. We refer to it as *low level* because it is meant to be a framework upon which we can define other higher-level syntaxes that offer more convenience. But, the higher-level syntaxes can always be lowered into the low-level, local syntax.

The fundamental idea for the local syntax is to extend standard data clauses with optional nests of clauses. For a data clause of the form

clause(var-list)

we extend it to support a nested form

top-clause(var-list)[::{ nest-clause-list }]

where *top-clause* is an ordinary data clause, *var-list* is a list of objects with the same user-defined type, and *nest-clause-list* is a list of data clauses applied to members of that user defined type. Informally, the nesting syntax allows a user to specify data clauses for the members of a user-defined type. The nesting generalizes to arbitrary depth, supporting user-defined types that have members of user-defined type, and it applies similarly to `update` directives.

Figure 12 shows how the low-level syntax can express full deep-copy, selective direction, selective member, and mutable deep-copy for a user-defined type with two pointer members. For name resolution, the nest is evaluated as though it is in the namespace of the user-defined type; this allows placing pointer members directly in nested data clauses, and it allows a shape expression to directly reference other member variables. In this example the user-defined type contains an integer member that specifies the shape of the member pointers. When a shaped member pointer appears in a nest, it implies an *attach* operation at the start of the data region and a *detach* operation at the end of the data region.

The nested syntax allows specifying the top-level object and every sub-object in a separate data clause. So, full deep-copy is achieved by specifying the `copy` clause for the top-level object and all nested sub-objects. Selective-direction deep-copy just requires using different data clauses for the top-level object and various sub-objects. In the example,

```
struct Deep {
 int size;
 double scalar;
 double* A; /* A[0:size] */
 double* B; /* B[0:size] */
};

void deep_copy(struct Deep* p, int n) {
 #pragma acc data copy(p[0:n])::{ copy(A[0:size],
      B[0:size]) }
 {
  /* p is available for use on device */
 }
}

void selective_direction(struct Deep* p, int n) {
 #pragma acc data copy(p[0:n])::{ copyout(A[0:size])
      copyin(B[0:size]) }
 {
  /* p is available for use on device */
  /* p[:].A[:] = p[:].B[:]*p[:].scalar */
 }
}

void selective_member(struct Deep* p, int n) {
 #pragma acc data copyin(p[0:n])::{ copy(A[0:size]) }
 {
  mutable_copy(p, n);
 }
}

void mutable_copy(struct Deep* p, int n) {
 #pragma acc data present(p[0:n])::{ copy(B[0:size]) }
 {
  /* A and B are both now available on the device */
 }
}
```

Figure 12. Local nested syntax

the top-level object and sub-object `B` appear in `copyin` clauses and sub-object `A` appears in a `copyout` clause. Similarly, selective-member deep-copy can be achieved by not specifying a data clause for a member that should not be transferred. In the example, the top-level object appears in a `copy` clause, sub-object `A` appears in a `copy` clause, and sub-object `B` is not specified in any clause. The local syntax is fully explicit, so allocations and transfers only occur for members that appear in clauses. Finally, mutable deep-copy is just a more general form of selective-direction deep-copy where at least one clause is a `present` clause. In the example the top-level object appears in a `present` clause and sub-object `B` appears in a `copy` clause. This directive triggers a lookup on the top-level object to ensure it is indeed present; then sub-object `B` is allocated and copied to the device, and it is attached to the existing top-level object. At the end of the data region sub-object `B` is transferred back to the host, detached from the top-level object, and deleted. The top-level object is left on the device, since it appeared in a `present` clause.

1) Implementation details: We made several design decisions that simplify the use of our syntax by increasing the complexity of an implementation. This section will describe these decisions and the impact they have on implementations. In both cases we made a tradeoff to favor improving a user's experience over reducing implementation complexity.

Automatic alias resolution : OpenACC does not currently define a processing order for the data clauses on a directive. However, deep copy in the presence of aliasing introduces an ordering constraint. If a data structure contains no aliases, then the order in which that data structure is traversed and replicated on the device does not matter. Further, traversal order does not matter for data structures with only *external aliases*, which are aliases to objects that are already present prior to the current transfer directive. But, order does matter for data structures with *internal aliases*, which are aliases to objects within the same data structure. All internal aliases must be resolved last, to ensure that the aliased object is created prior to translating an alias. The most complex case is when two or more conditional aliases (i.e., `present_or` aliasing) appear in the same data structure. In this case the conditional clause that is processed first will create the object, and all other aliased conditional clauses will find the existing object and simply use that. So, the largest conditional alias must be processed first to avoid under-allocating the object.

Because the ordering constraints can be so complex, it is impossible to define a static order that will address all situations. Allowing the user to specify a traversal order might work, but providing mechanisms to cover all the cases becomes very awkward (e.g., depth-first vs. breadth-first, pre-order vs. post-order, and sibling order). Moreover, applications can exhibit complex aliasing properties that can be difficult for even the application developers themselves to describe. As a result, we chose to place the ordering burden on the implementation rather than the user. This means that a deep-copy implementation must perform automatic alias resolution and traversal ordering.

Deciding the proper traversal order requires multiple passes over the data structure to detect internal aliases. If there are no aliases, or only external aliases, then any traversal order will suffice. If there are internal aliases, then an implementation must traverse the objects in an order that resolves allocation conflicts and ensures all aliases are present when traversed. In essence, the deep-copy implementation must defer all allocations until the entire data structure has been considered. At that point it can automatically perform the allocations and translations in the proper order.

Although the requirement for automatic alias resolution complicates implementing our proposed deep-copy syntax, we believe it is better to hide the complexity in the implementation than to expose it to users. This decision allows users to express deep-copy in a natural manner, without regard for traversal ordering or aliasing considerations. In other words, users will get the behavior they expect without concern for how it is achieved.

Translated pointer table: Raw data, which is interpreted the same on both host and device, can be copied between host and device memory using ordinary memory

transfers. But when an object contains an attached pointer, as is the case for deep copy, it cannot be simply copied between host and device memory – it must be properly translated. For data regions that employ deep copy, all necessary pointer translation is implied by the deep-copy operation. But, update operations differ. Consider the case where an object is deep-copied in a data region but shallow-copied in an `update` directive. If implemented naively, the shallow `update` will transfer the entire top-level object, including translated pointer members. This will result in a device pointer on the host or a host pointer on the device, depending on the direction of the transfer, and the sub-objects will no longer be accessible through the top-level member pointers.

To address this problem we're proposing a new clause, `maintain`, which is allowed in an `update` nest. The `maintain` clause specifies that a member should not be modified by an update. So, a shallow update can be performed on an object with translated pointer members if the pointer members appear in nested `maintain` clauses. This clause essentially instructs an implementation to transfer around any specified members, avoiding the translated pointer problem altogether.

The responsibility of specifying a `maintain` clause where necessary could be relegated to the programmer; but we believe this requirement would place a non-trivial burden on the programmer. Instead, we again decided to ease programmer burden by shifting responsibility to the implementation. Thus, an implementation must track all translated pointers and treat them as though they appear in `maintain` clauses for `update` directives. This allows programmers to perform shallow updates of variables with member pointers without concern for whether or not those member pointers have been translated.

The main concern with requiring an implementation to track translated pointers is overhead, both time and space. OpenACC already requires that an implementation maintain a *present table* to map host address ranges to corresponding device address ranges. The present table requires one entry per distinct object or array. The translated pointer table, on the other hand, requires one entry per translated pointer.

For data structures with no aliasing, every object will have exactly one pointer targeting it. Thus, the sizes of the present table and the translated pointer table will both be $O(n)$, where n is the number of objects. For this case, the overhead of managing the translated pointer table is only a constant factor more than managing the present table.

For data structures with extreme aliasing, however, the translated pointer table can be significantly larger than the present table. Consider a sparse-matrix representation with two arrays, a flat data array and an array of row pointers that each target some location in the data array. The present table will only contain two entries, one for each array. But, the translated pointer table will contain an entry for each element in the row-pointer array. In this case, the size of the present table is $O(n)$, where n is the number of objects, while the size of the translated pointer table is $O(n * m)$, where m is the maximum number of pointers per object. In theory, an application with many objects and extensive aliasing could begin to experience scaling problems.

But despite poor worst-case scaling, we expect that for most applications the translated pointer table will incur little overhead, especially when compared to the cost of transferring memory between host and device. Further, the translated pointer table can be optimized in several ways. First, it can be represented with a hash table, since every entry is a single fixed-size address (in contrast to the present table, which stores address ranges). Second, each entry in the present table could store its own translated pointer table for all pointers in that range. This does not reduce the total size of the translated pointer table, but it does make accesses more efficient by leveraging locality in the present table.

In short, requiring an implementation to maintain a translated pointer table certainly increases the complexity of an implementation. However, the end goal is to improve user experience. With an implementation that is aware of all translated pointers, users can update variables without concern for whether or not they contain translated pointers. In the end, users will be much less likely to unexpectedly find host or device pointers in the wrong place.

B. Transfer policies

The advantage of a local deep-copy syntax, one that is specified completely at the point of transfer, is that it allows every data region and update to express a custom deep-copy behavior. But limiting the syntax to only local specifications can be problematic. First, a purely local syntax inhibits reuse – a deep-copy behavior that applies to many different variables in many different locations must be fully specified for each variable and location. Second, and more importantly, a purely local syntax breaks object encapsulation, which is an important property for user-defined types, particularly for programs that follow an object-oriented paradigm. Consider a user-defined type with private pointer members. Achieving deep copy with a local syntax requires exposing those private pointer members to the code that performs the transfer. Exposing private members is not so much a permission problem as it is a philosophical one – client code of a user-defined type should never be required to understand or even know the names of its private members. Instead, a properly encapsulated user-defined type should have corresponding shaping and deep-copy specifications that are provided by the author of the type. Then, any code using that type need not know anything about the type's internals or how to invoke deep-copy (or even whether deep-copy is necessary). Hence, we propose a new `acc policy` directive as a mechanism for specifying default and named *transfer policies* for user-defined types.

```
struct Deep {
 int size;
 double scalar;
 double* A; /* A[0:size] */
 double* B; /* B[0:size] */
 #pragma acc policy(shape) shape(A[0:size], B[0:size])
 #pragma acc policy("deep") inherit(*)
 #pragma acc policy("sel_dir") copyout(A[*]) copyin(B[*])
 #pragma acc policy("sel_mem") copy(A[*])
 #pragma acc policy("mut_copy") copy(B[*])
};

void deep_copy(struct Deep* p, int n) {
 #pragma acc data copy(deep:p[0:n])
 {
  /* p is available for use on device */
 }
}

void selective_direction(struct Deep* p, int n) {
 #pragma acc data copyin(sel_dir:p[0:n])
 {
  /* p is available for use on device */
  /* p[:].A[:] = p[:].B[:]*p[:].scalar */
 }
}

void selective_member(struct Deep* p, int n) {
 #pragma acc data copy(sel_mem:p[0:n])
 {
  mutable_copy(p, n);
 }
}

void mutable_copy(struct Deep* p, int n) {
 #pragma acc data present(mut_copy:p[0:n])
 {
  /* A and B are both now available on the device */
 }
}
```

Figure 13. Named transfer policies

Figure 13 illustrates how transfer policies are specified with the `acc policy` directive. The special `shape` policy is used to define default shapes for member pointers; shapes defined in this manner are for convenience, as they are made available to all other policies and local nest syntax. In the example, members `A` and `B` appear in a `shape` policy.

The example also has a named policy, `"deep"`, that uses the new `inherit` clause. The `inherit` clause only applies to members, and it indicates that a member will behave as though it appeared in the same clause as the parent object. The "`*`" argument to the `inherit` clause expands to include all shaped indirect members. To illustrate, the data clause that invokes this `"deep"` policy is `copy( deep:p[0:n] )`. The named policy lowers into low-level syntax as `copy( deep:p[0:n] )::{ inherit(*) }` The `inherit` clause pulls in the default `shape` policy, lowering to `copy( deep:p[0:n] )::{ copy( A[0:size], B[0:size]) }`. Through these lowering steps, it becomes clear that the `"deep"` policy specifies a full deep-copy. Using the same mechanisms, the `"sel_dir"`, `"sel_mem"`, and `"mut_copy"` policies define behavior for selective-direction, selective-member, and mutable deep-copy. The `[*]` syntax indicates

```
template<typename T>
class my_vector {
private:
  T* _begin;
  T* _end_data;
  T* _end_storage;

public:
  T& operator[](int index) {
    return _begin[index];
  }
  const T& operator[](int index) const {
    return _begin[index];
  }
#pragma acc policy(shape) shape(                          \
  _begin[0:(((_end_data - _begin)/sizeof(T)) - 1)])
#pragma acc policy(data) inherit(_begin[*])  \
  present(_end_data[@_begin])                \
  present(_end_storage[@_begin])
#pragma acc policy(update) update(_begin[*])
};

class Data {
private:
  my_vector<double> A;
  my_vector<double> B;
  my_vector<int>    C;
  my_vector<int>    Other;

#pragma acc policy(shape) shallow(Other)
#pragma acc policy(data) inherit(*)
#pragma acc policy(update) update(*)
};

// Examples

my_vector<double> vec1,vec2;

// implicit policies let my_vector be treated like a
// simple variable
#pragma acc data copyin(vec1) copy(vec2)

#pragma acc update self(vec1)

Data dat1;

// equivalent to copy(dat1)::{ copy(A,B,C) }
#pragma acc data copy(dat1)

// override policy when you need to, copyin(B) don't
// copy A, copy(C)
#pragma acc data copyin(dat1)::{ shallow(A) copy(C) }
```

Figure 14. Implicit transfer policies

that an array shape is specified in a `shape` policy.

In addition to named policies, users may define implicit `data` and `update` transfer policies. These special policies are applied in all data clauses that do not specify an explicit policy name. This feature allows an author or a user-defined type to define default deep-copy behavior that will apply to all variables of that type.

Figure 14 illustrates how implicit policies can be used to encapsulate deep-copy behavior in a simplified C++ `std::vector` class. The vector is a template type, where the template parameter defines the vector element type. The class has three pointers, designating the beginning of the data, the end of the data, and the end of the reserved storage. The two end pointers alias the memory targeted by the begin pointer, although the end of storage pointer actually points

to one-past the end of valid memory.

The `shape` policy encapsulates most of the complexity of this example. The begin pointer is shaped with a size computed by subtracting the begin pointer from the end pointer. This policy chooses to transfer the active data rather than reserved storage, although we could define another named policy to achieve both behaviors.

Both the `data` and `update` policies inherit the previously defined shapes, using them in `inherit` and `update` clauses. These two policies define default deep-copy behavior for data regions and updates. This allows variables of type `my_vector` to appear directly in data clauses, like `copyin(vec1)` or `copy(vec2)`. For the `data` policy, the two end pointers appear in `present` clauses and are shaped with the new @ syntax, which indicates that they are aliases with respect to the begin pointer. Thus, these pointers will be translated when transferring an object of this type.

Further, the class `Data` contains direct members of type `my_vector`. These members will behave as though they have an implicit shape. So, they may be deep-copied by explicitly specifying them in a data clause or implicitly specifying them with a data clause using the `*` argument. The default shape policy for the `Data` class specifies `shallow(Other)`, which disables the default policies for that member; so, the `inherit(*)` and `update(*)` clauses only specify deep-copy for members `A`, `B`, and `C`.

With implicit `data` and `update` policies in the `Data` class, it may be specified directly in a data clause like `copy(dat1)`. Doing so is equivalent to specifying `copy(dat1)::{ copy(A,B,C) }`. Finally, an implicit policy can always be overridden with local syntax, as shown at the end of the example.

V. Complications and future work

In addition to the primary challenges described in Section II, there are many secondary complications related to implementing deep-copy. These issues are certainly not an absolute road-block for implementations, but they do suggest that initial versions will likely impose several key restrictions.

Modifying pointers in data regions: A major assumption throughout this paper is that the *structure* of a deep data structure remains fixed throughout the entire data region that places it on the device. That is, the pointer relationships between objects in a data structure may not change. This limitation is similar to requiring the shape of a flat object to remain fixed throughout the data region that places it on the device. For example, if a variable is transferred to the device with `copy(x[0:n])`, then the variable `n` may not be altered prior to the end of the data region. Relaxing this constraint complicates the transfer at the end of the data region – should the transfer honor the original or new value of `n`? Likewise, altering a pointer relationship within a data region presents similar ambiguities – should the final deep-transfer and deep-free honor the original or new pointer relationship? Honoring the new pointer could result in complex memory management issues, such as freeing the same pointer more than once or failing to free a pointer at all (i.e., orphaning an object). Given these challenges, we believe at least initial support for deep-copy in OpenACC will prohibit modifying pointers within data regions. Future revisions might be able to relax this limitation in various ways. For example, allowing simple pointer aliases to be modified is relatively straightforward, since it just requires translating the pointer upon transfer. Alternatively, requiring a user to notify an implementation when a pointer has changed (perhaps through an explicit pointer update directive) might reduce the complexity of the problem.

Memory management on the device: OpenACC defines a master-slave model where a thread on the host initiates all actions on the device. Conceptually, all device memory management is initiated by the host, and all device objects are created to mirror corresponding host objects.[4] It is unclear what behavior should result if a compute region attempts to allocate or free memory, particularly if that memory is previously or subsequently involved in a data transfer between host and device. Presumably a compute region could manage memory that is only active for the lifetime of that compute region – in this case, the memory would be allocated after the start of the compute region and deallocated before the end of the compute region, and there would be no need to allocate corresponding host memory. But, what if a compute region attempts to deallocate memory that mirrors a host object? Should an implementation detect this situation, perhaps also deallocating the host object at the end of the data region? Similarly, if pointer reassociation is allowed on the device, then a compute region could allocate an object and store the pointer in a deep data structure that will be copied back to the host at the end of the data region. Should an implementation detect this and allocate a corresponding host object at the end of the data region? Both of these cases deviate from the expected paradigm, where all device memory management is initiated by the host, and all device objects mirror host objects.

This question of device memory management arises for standard C++ containers, such as `std::vector`. Transferring a `vector` requires deep copy, but supporting a resize operation on the device requires much more – it requires allowing pointers to change and allowing memory management on the device. Since both of these capabilities introduce significant complications, it seems likely that initial versions of deep-copy will likely prohibit resizing standard containers. Application developers with whom we've been working seem willing to accept this limitation, as long as containers

[4]Technically the `device_resident` clause allows an implementation to elide a host copy of an object; but, the device copy is still a logical mirror of the host copy, even if the host copy hasn't been created.

may be resized between data regions.

A possible workaround for `std::vector` is to reserve additional space prior to a data region. Then the deep-copy policy would need to transfer the reserved space, not just the occupied space, allowing the compute region to append into the pre-reserved space.

Conditional deep traversal: The semantics of a data-clause nest require that it always be traversed, even for conditional data clauses. So, for a `present_or` clause, the nest is traversed regardless of the outcome of the present test. This traversal is needed to support top-down mutable deep-copy, where a conditional top-level clause contains a conditional nested clause. Even if both objects are present, the nest must be traversed to attach the two objects.

There are subtle implications of this design decision. First, it is impossible to express a conditional, shallow present test for a deep-copy nest where the top-level object appears in a conditional clause. (An unconditional, shallow present test can be achieved by using a `present` clause without a nest.) Second, it is not possible to handle recursive, cyclic data structures, since a nest is always processed regardless of the outcome of a conditional clause present test.

We considered extending the nest syntax with `ifpresent` and `ifnotpresent` clauses, which would make nest processing conditional based on the outcome of the present test for the parent clause. These clauses address the problems described above, but we decided to defer them because they complicate the syntax. As a generalization, we could also consider a generic `if` clause.

C++ semantics: C++ exposes object creation, deletion, and assignment through explicit constructors, destructors and operators. However, OpenACC does not specify whether transferring a C++ object will invoke these standard routines. The essential question is whether a device copy of C++ object is truly a new object, or whether it is a temporary clone of the original host object. If we view it as a new object, then creating and transferring such an object should invoke the appropriate constructors, destructors and operators. The advantage of this approach is that it exposes the transfer mechanism to programmers, potentially giving them a way to control an object's transfer behavior. The disadvantage is that the behavior will differ for systems with unified memory, where an actual transfer is not required.

On the other hand, if we view a device object as a temporary relocation of the original host object, then creating and transferring an object should not imply language-level creation and copying of objects. In this case, the two objects are logically the same object, they just appear in different physical and temporal localities. The authors of this paper tend to prefer this view, where object transfer is not subject to ordinary language semantics.

Finally, there may be a middle ground where we define special member functions to be called when an object is transferred. This solution still views host and device objects as a single logical object, but it exposes transfer operations to the programmer.

Pointers in unions: Our deep-copy proposal does not provide a direct mechanism for scoping a pointer that appears in a C/C++ union. In such cases the pointer may or may not be valid, depending on the dynamic manner in which the union is used. The best option with our current proposal is to shape the pointer with a conditional size, where the size evaluates to 0 when the pointer is not valid. If we extend the syntax to support conditional clauses, then the pointer could just be shaped conditionally. Fortunately, the OpenACC user community has not expressed any need for supporting pointers in unions.

Unstructured data regions: For unstructured data directives, do `enter` and `exit` deep-copy policies have to match? This problem also exists in the absence of deep-copy, where an `enter` and `exit` directive use different shapes. The main complication for deep-copy is that pointer traversal and translation could differ for the two directives. However, the attachment table can at least catch errors. For example, if a shallow-`copyout` follows a deep-`copyin` then an implementation could detect that an object being deleted contains attached pointers. It could detach the sub-objects, and optionally delete those sub-objects if the reference count drops to 0. Similarly, if a deep-`copyout` follows a shallow-`copyin` then an implementation could issue an error that the object is not attached.

Untranslated pointers: How should pointers that aren't translated be handled? Should they remain as host pointers? Should they be set to null? Several users have indicated that setting them to null could be a useful debugging aid, or even a mechanism to check for presence of a sub-object. The disadvantage of setting them to null is that it causes the translated pointer table to grow larger. Also, behavior differs for systems with unified memory. Thus, the easiest solution is to make it undefined behavior to access an untranslated pointer member. An implementation could then provide various debug options to help users detect problems.

C++ templates: we need to be able to use these directives inside of "templatized" types (possibly even declaring policies outside of them); what if the template type parameter is a pointer? we can currently handle a type parameter that is a user-defined type with an implicit policy, but not for a raw pointer; might need to use template specialization, or policy specialization based on template type (e.g., "type_is_pointer"); might want a way to declare a named policy for a non-struct pointer, which could then be specified as the "base type policy" for a template class

Polymorphism: Polymorphic types present a unique challenge to deep-copy. Namely, the exact size of a polymorphic type may not be known statically. Thus, polymorphism is not supported by our current proposal.

The most natural way to support polymorphic types is

to support *virtual policies*, similar to virtual function calls. A base class defines a virtual policy, and all sub-classes override that policy. At runtime, a virtual policy is resolved to that of the proper concrete type.

However, polymorphism presents additional challenges in OpenACC. For example, virtual function tables must be translated to target the device copy of each member function. Given these additional complexities, full support for polymorphism will likely take some time. Fortunately, high-performance codes often avoid polymorphism, so this functionality is not urgently needed.

Recursive data structures: Our proposed local syntax doesn't support recursive data structures, since it requires explicitly specifying a nest for every sub-object. Instead, defining a transfer policy naturally supports recursive types. Even so, cyclic data structures pose a termination problem discussed previously in conditional deep traversal.

Asynchronous deep-copy: Achieving high performance with OpenACC often requires asynchronous transfers and computation. In theory, a deep-copy transfer is perfectly capable of running asynchronously. But, performing pointer translations asynchronously adds an additional level of complexity. Fortunately, an initial implementation can legally synchronize to ensure correctness, later adding true asynchronous deep-copy in future versions.

Scaling: In the absence of deep-copy, every variable specified in a data clause triggers a single memory allocation and transfer. Even large, dynamically-sized arrays corresponded to a single block of contiguous memory. This implies that the number of independent objects that an implementation must manage is relatively small, bounded by the number of variables appearing in data clauses in a program. In contrast, deep-copy allows a single variable to trigger allocation and transfer of an arbitrary number of disjoint objects. This is a fundamental paradigm shift, moving from few large objects to many small objects, and it has direct scaling implications for OpenACC implementations. Since an implementation must track all host and device object mappings, a implementation with poor scaling will quickly become apparent for large deep-copy use cases. To reduce such scaling problems, programmers could employ aggregate allocation strategies, such as allocating many small objects with a single large allocation. This allows transferring a large group of objects with a single contiguous transfer. Implementations could automate similar strategies when applicable.

VI. Conclusion

OpenACC has received much attention in the high-performance computing community recently, and it is viewed by many people as a notable step forward in easing the difficulty of programming heterogeneous accelerator systems. The promises are lofty: a directive-based approach allows a single source-code base to target both homogeneous and heterogeneous systems; a descriptive rather than prescriptive set of directives facilitates performance portability by giving an implementation the flexibility to optimize and efficiently map a program onto different accelerator targets; and, the abstraction of disjoint host and device memory, coupled with compiler and runtime assistance in managing device memory, frees a programmer from the tedious details of manually managing device memory, a task that was historically synonymous with accelerator programming. However, as excitement around the initial success of OpenACC begins to fade, it becomes clear that lack of deep-copy support is the single largest impediment to porting a wider range of interesting data structures and applications to OpenACC. With no better option, users are forced into manually managing device memory and rewriting large portions of their application code – an outcome that runs counter to the entire purpose of a directive-based programming model.

The potential benefit of providing a directive-based solution for deep copy is great, as it would extend the usefulness of OpenACC to a much larger set of applications. But in general it is a very challenging problem, as illustrated in this paper. Even so, we believe the problem is solvable, especially with some reasonable assumptions and limitations that we've outlined in the paper. Our proposed low-level, local syntax is relatively compact and concise, yet it is capable of expressing the deep-copy capabilities that real applications need: selective member, selective direction, and mutable deep-copy. Moreover, our member shape and policy syntax builds naturally upon the local syntax and facilitates good software engineering techniques such as policy encapsulation and reuse. With this new syntax the deep-copy requirements for many applications can be expressed solely through compiler directives, without any additional code modification, allowing a compiler and runtime to automatically handle the tedious details of allocating and transferring complex data structures between host and device memory.

References

[1] *The OpenACC Application Programming Interface, v2.0*, OpenACC.org, Aug. 2013. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf

[2] *CUDA Toolkit Documentation*, Nvidia. [Online]. Available: http://docs.nvidia.com/cuda/index.html

[3] *Cray Fortran Reference Manual S–3901–82*, Cray Inc., Sep. 2013. [Online]. Available: http://docs.cray.com/books/S-3901-82/S-3901-82.pdf

[4] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, Knoxville, TN, USA, Sep. 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[5] *OpenACC.examples*, Man Page, Cray Inc., Sep. 2013. [Online]. Available: http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=;f=man/xt_prgdirm/82/cat7/openacc.examples.7.html