# Automatic Copying of Pointer-Based Data Structures

Tong Chen, Zehra Sura, and Hyojin Sung

IBM T.J. Watson Research Center, New York, USA,
{chentong,zsura,hsung}@us.ibm.com

**Abstract.** In systems with multiple memories, software may need to explicitly copy data from one memory location to another. This copying is required to enable access or to unlock performance, and it is especially important in heterogeneous systems. When the data includes pointers to other data, the copying process has to recursively follow the pointers to perform a deep copy of the entire data structure. It is tedious and error-prone to require users to manually program the deep copy code for each pointer-based data structure used. Instead, a compiler and runtime system can automatically handle deep copies if it can identify pointers in the data, and can determine the size and type of data pointed to by each pointer. This is possible if the language provides reflection capabilities, or uses smart pointers that encapsulate this information, e.g. Fortran pointers that intrinsically include dope vectors to describe the data pointed to. In this paper, we describe our implementation of automatic deep copy in a Fortran compiler targeting a heterogeneous system with GPUs. We measure the runtime overheads of the deep copies, propose techniques to reduce this overhead, and evaluate the efficacy of these techniques.

**Keywords:** parallel computing, heterogeneous systems, compilers, memory

## 1 Introduction

Massive parallelism and heterogeneity are prevalent in current systems designed for compute-intensive applications. These systems typically include multiple distributed memories, and software may need to explicitly copy data from one memory location to another. In some cases, this copying is necessary for certain processors in the system to be able to access the corresponding data. For example, in a system with host processors and GPU accelerators connected via an interconnect (e.g. PCIe), the system-wide memory and the on-chip GPU memory have separate address spaces. Host processors can directly refer to addresses in the system-wide memory, but the GPU processors can only refer to addresses in the on-chip GPU memory. Any program data operated on by the GPU has to be explicitly transferred to/from the system-wide memory. In other cases, all the processors in the system share a global address space, but because of non-uniform memory access times, it may still be worthwhile to copy data between different memory locations to combat performance loss due to NUMA effects.

For application codes that use pointer-based data structures, the data to be copied includes pointers to other data, and the copying process has to recursively follow the pointers to perform a *deep copy* of the entire data structure. Further, pointer address values in the copied data have to be fixed to refer to addresses in the copied version of

the data structure. It is tedious and error-prone to require users to manually program the deep copy code for each pointer-based data structure. Instead, a compiler and runtime system can automatically handle deep copies if it can identify pointers in the data, and can determine the size and type of data pointed to by each pointer. This is possible if the language provides reflection capabilities, or uses smart pointers that encapsulate this information, e.g. Fortran pointers that intrinsically include dope vectors to describe the data pointed to.

While our ideas are generally applicable to distributed memory systems, in this paper we focus on a CPU-GPU system with a host IBM POWER8 processor connected to an NVIDIA Kepler GPU via PCIe. Currently, the most common method used to program data transfers in such a system is to use the CUDA API[15] which provides runtime library calls for memory management and data transfers. However, this is a low-level API, and using it to manually program data copies can adversely affect productivity of software development.

An alternative method is to use CUDA Unified Memory[9], which provides a shared address space abstraction across the host processor and the GPU, with the underlying implementation transparently and automatically handling all data copies. Unified Memory is very easy to use from the programmer's perspective, but it can degrade performance for some applications since it is a uniform (one-size-fits-all) solution that works at page-based granularity and cannot be customized per application.

Yet another method for programming data transfers in a CPU-GPU system is to use a directive-based approach, such as OpenACC[17] or OpenMP[3] with accelerator support. These provide high-level annotations that the programmer can insert at appropriate points in the code to identify data that will be accessed on the GPU. The OpenACC/OpenMP implementation then takes care of performing data copies when necessary.This implementation not only performs data transfers, but is also responsible for GPU memory allocation/de-allocation, and for tracking data items that have been previously copied. The directive-based approach has the advantage of allowing application-specific optimization while also alleviating the tedium of programming to a low-level API. However, the OpenACC and OpenMP standards currently do not support deep copy for pointer-based data. Many applications include pointer-based data structures, and to use OpenACC/OpenMP for such applications, programmers must either devolve to using low-level APIs for copying their data, or they must re-structure program data so that deep copy is not needed. The latter may involve major code changes and may not be feasible. While the standards are evolving and trying to address these issues, the deep copy problem is tricky to solve, in part because OpenACC/OpenMP are geared towards high performance computing and are sensitive to runtime overheads introduced due to specification of the standards.

In this work, we explored the design and performance implications of supporting deep copy semantics in a directive-based programming model for Fortran. Our system integrates components at three levels:

1. Language features: In Fortran, implementing some language features (e.g. dynamic array sections) makes it necessary for the executable code to be able to store and access extra information for pointer fields and variables. The format of this information is implementation dependent and is referred to as a dope vector. There is a dope vector associated with each pointer, and the information stored in dope vectors

can be accessed by runtime library code. Also, Fortran does not allow indiscriminate pointer casting or pointer arithmetic, which simplifies pointer handling by an automatic system.
2. Compiler analysis: For all types used in an application (intrinsic or user-defined types), information about the size and layout of each type is extracted in the compiler and made available to the runtime system.
3. Runtime system: Runtime library functions implement the code for data transfers, making use of dope vectors and compiler generated information to perform pointer traversals for deep copy.

We inserted OpenMP *map* clauses in Fortran program codes to identify data to be copied to or from the GPU memory. We modified our Fortran OpenMP compiler and runtime implementation to automatically support deep copy for all pointer-based data in the *map* clauses. Since Fortran pointers include dope vectors that describe the data being pointed to, our system has ready access to the information needed to support deep copy.

Contributions of this paper are as follows:

- We describe the design and implementation of our compiler and runtime support for automatically copying pointer-based data structures in Fortran OpenMP codes targeting a CPU-GPU system. Our algorithms include support for recursive data structures and cyclic pointer traversals (Section 2).
- We introduce techniques that can be applied to reduce the runtime overhead of deep copy (Section 3).
- We collect experimental data to measure the runtime overheads of our deep copy implementation, and evaluate the effectiveness of the techniques proposed to mitigate this overhead (Section 4).

## 2 Design and Implementation

Figure 1 shows a code snippet for declaring a simple pointer-based list data structure, and using OpenMP to copy and process the list on the GPU. Lines 7-9 form an OpenMP *target* region that is to be executed on the GPU. The OpenMP *map* clause on Line 7 is used to identify data to be copied to and from GPU memory. The map clause can be used with multiple options, for example it can specify that data only be mapped to the GPU, or only be mapped from the GPU. The default behaviour for mapping a data item is the following:

- On entry to a target region, if there is no copy of the data item in GPU memory, allocate it and transfer data to GPU memory.
- On exit from a target region, if this is the end of the lifetime of the data item, transfer data from the GPU copy to the host copy, and de-allocate GPU memory. The OpenMP specification includes rules that a runtime implementation has to use to keep track of the lifetimes of mapped data items.

### 2.1 Compilation

In our system, the compiler performs two functions relevant to data mapping. First, it inserts calls to the OpenMP runtime library to handle data copying for each data

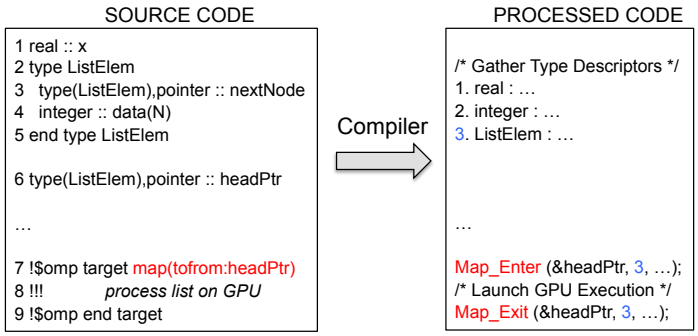


Fig. 1: Example to Illustrate Compiler Actions

item specified in a map clause. These calls, Map_Enter and Map_Exit, are illustrated in Figure 1 and described in Sections 2.4 and 2.5. Second, it collects high-level type information and passes it to the runtime. In the example in Figure 1, information for 3 types is collected: real, integer, and ListElem. The format used for passing type information is described in Section 2.2. The compiler can statically determine if a data item requires deep copy (i.e. if it is of pointer type, or if it contains pointer types), and if so, it passes the corresponding runtime type descriptor index as a parameter to the OpenMP library call inserted for the map. The runtime then uses this type descriptor information to recursively traverse the entire data structure and perform deep copy. In our design, the user can control when deep copy is performed by using an extension of OpenMP map-types to override the automatic deep copy behavior in specific map instances.

**Dope Vectors** Information in a pointer variable typically contains only the address of the data pointed to. However, a Fortran pointer variable carries more information, as illustrated in Figure 2. This information, collectively called the dope vector, is implementation dependent and may include the data address, a flag to indicate if the pointer is associated with valid data, the size of data, and shape of the data for array types. The shape information includes number of dimensions and bounds for each dimension. In our compiler, we use the existing format for dope vectors as-is. Fortran pointers are typed, i.e. a given pointer variable can only be associated with data of a matching type. The size of the dope vector can vary depending on its associated data type, but this size is known statically at compile time. The size of array data and bounds of array dimensions may be dynamically determined and recorded at runtime in the corresponding fields of the dope vector. Our system correctly handles copying of arrays with dynamic lengths. Also, our compiler processes Fortran allocatable arrays and Fortran pointers to arrays in a similar manner, and we treat them uniformly in the copying implementation.

**Deep Copy** When copying a Fortran pointer between memories, both the dope vector and the data being pointed to have to be copied. Further, the address in the copied dope vector has to be updated to refer to the copied version of the data, as illustrated in Figure 3(a). The runtime keeps track of data already copied by recording the corresponding

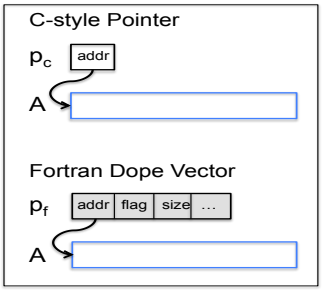

Fig. 2: Dope Vector

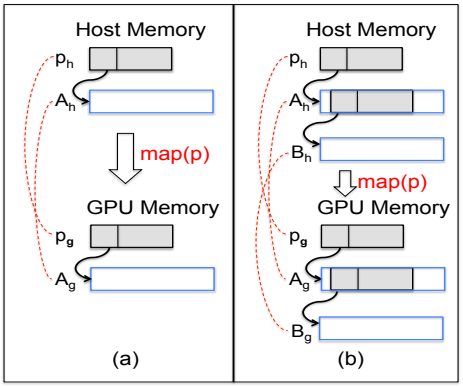

Fig. 3: Mapping Fortran Pointer-based Data

pair of dope vector addresses, and the corresponding pair of data addresses, shown by the dashed lines in the figure.

When performing a deep copy, the data structure has to be traversed by following pointers within the data being copied. For such pointers that are not the top-level pointers, the dope vector is contained within the data already copied over, as illustrated in Figure 3(b). In this case, only the data being pointed to has to be copied, and the address field in the dope vector has to be updated.

### 2.2 Runtime Type Descriptors

We introduced *runtime type descriptors* in our compiler and runtime system. To traverse the data structure for deep copy, the runtime has to be able to identify what parts of the data are pointer fields, and the type of data that these pointers refer to. The compiler has access to all type information for variables used in a compilation unit. It can collect the information required for traversals and pass it to the runtime by generating code to initialize runtime type descriptors on program start-up.

Figure 4 illustrates the format of the runtime type descriptor list. The index of an element in the list serves as an identifier for a data type (user-defined or otherwise) in the program code. There is an entry in the list for each type that contains pointer fields or that may be the target type associated with a pointer variable. A list entry is a type descriptor which is an integer value giving the size of the data type in bytes, followed by zero or more integer-triplets. Each triplet denotes a pointer field contained in the corresponding data type, and includes the following information:

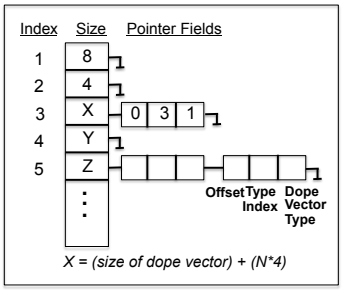

Fig. 4: Runtime Type Descriptors

1. Offset: length in bytes from the start of the data type to the pointer field.
2. Type ID: the index of the type descriptor list corresponding to the type of data pointed to by this pointer field.

3. Dope vector type: an identifier for the format of the dope vector corresponding to this pointer field. Our compiler uses different dope vector formats for scalar data versus arrays. For array types, each element of the array is traversed for deep copy.

In Figure 4, index 1 corresponds to real type, index 2 corresponds to integer type, and index 3 corresponds to the ListElem type in the example code snippet of Figure 1.

### 2.3 Assumptions

For automatic copying, we assume that the structure of the data is immutable during the time when multiple copies of the data exist. Specifically, this means that pointer fields within the data structure cannot change their value (both on the host, and on the GPU after the initial copy) during the lifetime of the mapped data. As a result, some application codes will not benefit from our automatic deep copy implementation and may need source code modification. However, there exists a large set of applications that will not be limited by this assumption. Note that the restriction applies only to pointers; other data fields may be freely modified.

Non-mutable pointers enable a low-overhead implementation of automatic deep copy. It may be possible to design algorithms that handle mutable data structures and work well in practice, but this is out of the scope of this paper.

### 2.4 Mapping Data on Target Entry

On entry to an OpenMP *target* region, the compiler generates host code to invoke a runtime library function for handling the data copy for each data item specified in a *map* clause. In our implementation we built upon an open-source OpenMP library[1], and modified it to support deep copy. Figure 5 shows the pseudocode for the runtime implementation. In this code, variable MapCount is used to track the lifetime of mapped data. We maintain MapCount for all data items reachable through deep copy traversals. We introduced variables globalMapID and MapID, which serve as timestamps to identify data items that have already been processed in a specific Map_Enter call. This allows our runtime to correctly handle cyclic pointer traversals in recursive data structures.

Figure 5(a), excluding the bold lines 8-11, 14, 18, and 19, is the existing code without support for deep copy. The Map_Enter function is invoked for each top-level data item to be copied. The runtime code keeps track of data that has been previously copied, maintaining a list of corresponding host and GPU addresses. It allocates GPU memory and transfers data for new copies. It also maintains a counter called MapCount for each host address to keep track of the lifetime of data copies. MapCount represents the number of top-level mapped variables that can reach a given address, either directly or through pointer traversals. It is used to automatically de-allocate GPU memory on exit from a *target* region for copies that can no longer be referenced.

The bold sections of Figure 5(a), together with the code in Figure 5(b), are our modifications for supporting deep copy. We introduced a variable, globalMapID, that is incremented on each call to Map_Enter and is unique to that instance of the call. We also introduced a MapID variable for each host address mapped, and set it to the

[1] Intel OpenMP Runtime Library: https://www.openmprtl.org

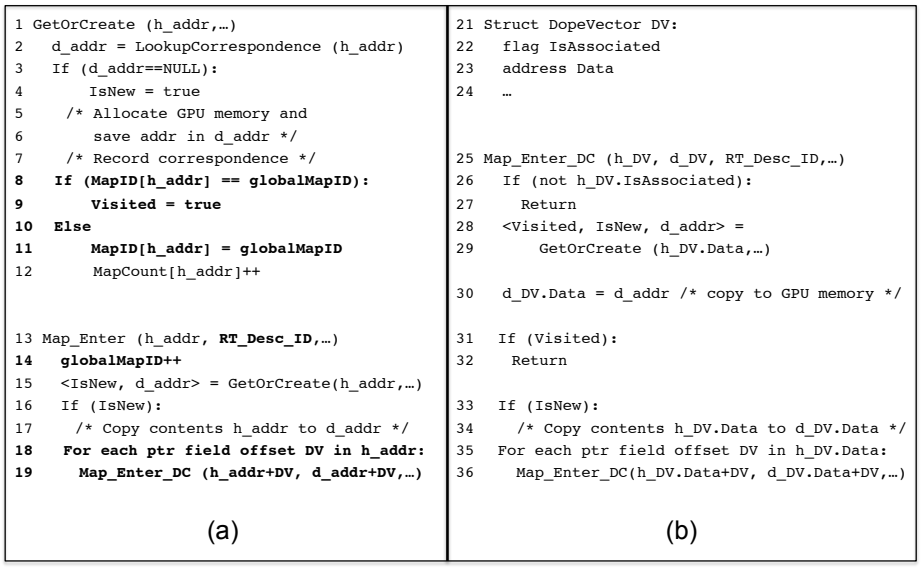


Fig. 5: Pseudocode for Copying Data on Target Entry

globalMapID value whenever a host address is processed as part of a Map_Enter call. Lines 8-11, 14, and 31-32 allow us to correctly handle recursive data structures when performing pointer traversals for deep copy. Lines 18-19 initiate the deep copy traversal by using the runtime type descriptor parameter to identify pointer fields in the data corresponding to the address being mapped. The Map_Enter_DC function is invoked for each of these pointer fields. This function is similar to the top-level Map_Enter function, except that it also checks if the pointer is associated with data (lines 26-27 that handle null pointers), fixes the pointer values in the GPU copy of the data (line 30), and handles recursive traversal (lines 35-36).

Note that the pseudocode in Figure 5 is simplified for clarity of presentation. The actual implementation is more complex because it includes optimizations as well as functionality to handle various *map* attributes that are part of the OpenMP specification. The deep copy part of the code also handles these attributes, propagating them in the recursive traversal. For aliasing of array sections, we impose the same restrictions as the current OpenMP standard, i.e. the first time an array is copied (mapped) in a *target* region, it must include all subsections of the array that will be subsequently mapped during the lifetime of the initial array copy. This allows us to reuse the existing logic in the runtime library to track corresponding addresses for host and GPU copies and avoid creating multiple copies of the same data.

### 2.5 Mapping Data on Target Exit

There is a runtime library function Map_Exit analogous to the Map_Enter function described in the previous section. On exit from an OpenMP *target* region, the compiler generates host code to invoke this function for each data item in *map* clauses associated

with the *target* region. Map_Exit uses the same globalMapID, MapID, and MapCount variables as Map_Enter, and it similarly traverses pointers for deep copy. The differences between the two functions are that:

- Map_Exit copies data in the reverse direction, from GPU memory to host memory.
- Map_Exit decrements MapCount instead of incrementing it.
- Map_Exit de-allocates GPU memory and deletes the correspondence when the MapCount for an address becomes zero.

## 3 Optimizations

The ease-of-use and productivity benefits of automatic deep copy have to be balanced with the runtime overhead of traversing data structures and performing multiple transfers corresponding to pointers in the data. In this section, we propose several techniques that can be used to reduce the runtime overhead.

### 3.1 Transfers to/from GPU Memory

When a user-defined data type contains a mix of pointer and non-pointer data, the pointer data has to be treated differently from the non-pointer data for the purpose of transfers to and from GPU memory. This is because the pointer address values in the GPU copy have to be fixed to point to data in GPU memory (refer to line 30 of the code in Figure 5). We describe 4 different techniques to perform data transfers of structures with a mix of pointer and non-pointer data. These techniques have different overheads depending on the number and contiguity of pointer fields and the size of data fields in the data type. In Figure 6, we illustrate the techniques using a simple example. In the figure, p and X represent host values for a pointer field and a data field, while $p_g$ and $X_g$ represent the corresponding GPU values. Dotted lines connect the same memory locations, and numbered circles represent the sequence of operations.

**1. Basic Version (BASE)**

**Copy to GPU Memory:** We first transfer the entire data structure to GPU memory. Then, for each pointer field, we transfer the GPU address value to the corresponding pointer field. Pointer fields are individually transferred only if they are associated.
**Copy from GPU Memory:** In this case, we cannot transfer the entire data structure to the host, since that will overwrite the original pointer address values on the host. Instead, we individually transfer each contiguous non-pointer data segment in the structure.

**2. Basic Version With Self-Managed Memory (BASE+)**

This is the same as the BASE version except that it uses self-managed GPU memory in the runtime. The CUDA library function, cudaMalloc, is used to allocate GPU memory. Repeatedly invoking this function during a deep copy can result in high overhead. In our implementation, we use a single call to allocate a large GPU memory space, and then self-manage this space in the runtime library to efficiently perform multiple smaller allocations and deallocations. All following versions (TCPY and PCPY) also use self-managed GPU memory.

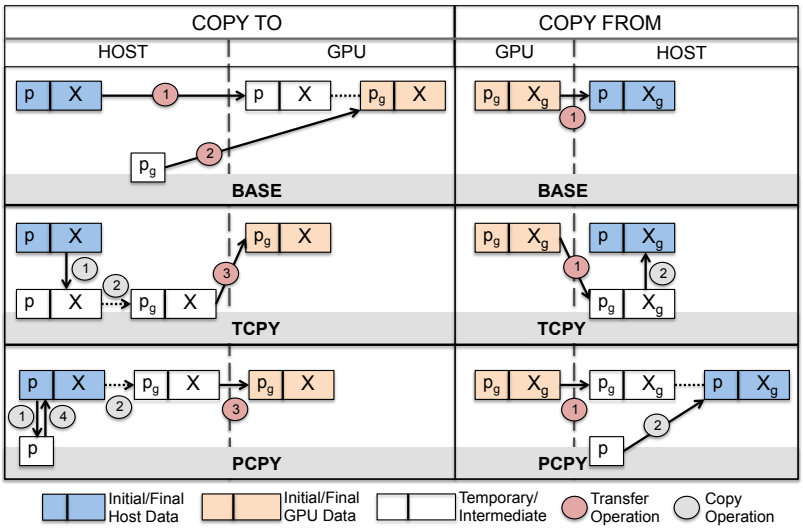


Fig. 6: Techniques to Optimize Pointer-based Data Transfers

**3. Version with Temporary Copies (TCPY)**

For this version, we first create a temporary copy of the data structure on the host.
**Copy to GPU Memory:** We overwrite the pointer address fields in the temporary copy with the corresponding GPU address values. Then we do a single transfer of the entire data structure from the temporary copy to GPU memory.
**Copy from GPU Memory:** We transfer the entire data structure from GPU memory to the temporary host copy. Then we copy only the non-pointer data from the temporary copy to the original data structure on the host.

**4. Version with Temporary Pointer Value Copies (PCPY)**

For this version, we assume that the pointers are not used on the host (due to accesses in multithreaded host code) during the processing of the map clause. This property can be determined by compiler analysis in some cases, or it can be provided by the user via program annotations.

We first allocate temporary space on the host, and for each pointer field, we copy the value of the host pointer to the temporary space.
**Copy to GPU Memory:** We update the pointer address values to corresponding GPU address values in-place in the host copy of the data. We then transfer the entire data structure to GPU memory. Finally, we restore the original pointer values in the host copy.
**Copy from GPU Memory:** We transfer the entire data structure from GPU memory to the host. Then for each pointer field, we copy the host address value of the pointer from temporary space to its original location.

For TCPY and PCPY, the runtime checks if a data item has any associated (non-null) pointers before it creates temporary copies on the host.

Table 1 gives the overheads associated with each technique in terms of number of transfers, size of data transferred, and size of temporary copies on the host. We assume S is the size of the data structure to be copied, DV is the size of a dope vector, and M is the number of pointer fields in the data structure. Note that the number of transfers for the copy-from case in the BASE versions depends on the contiguity of pointer fields in the layout of the data structure.

Table 1: Cost of Different Data Transfer Techniques

| | **Number of Transfers** | **Size of Transfers** | **Size of Host Copies** |
|---|---|---|---|
| BASE Copy To | 1+M | S+M*DV | 0 |
| BASE Copy From | varies | S-M*DV | 0 |
| TCPY | 1 | S | S-M*DV |
| PCPY | 1 | S | M*DV |

### 3.2 Other Optimizations

In this section, we discuss some other optimizations that can be applied based on information obtained from programmer annotations and/or sophisticated analysis.

**Structured Maps** In addition to the assumptions in Section 2.3, if it is known that data transfer directives are only associated with structured programming constructs[2], then the runtime overhead can be reduced. In this case, the globalMapID of Section 2.4 is used to track the level of the nesting structure by incrementing it on each Map_Enter call and decrementing it on each Map_Exit call. The MapID for an address is set to the current nesting level only when corresponding memory is newly allocated on the GPU in a Map_Enter or Map_Enter_DC call. That corresponding GPU memory is copied back/de-allocated at the end of the structured nesting level (i.e. in the first Map_Exit call that decrements the globalMapID to a value less than the MapID for the address). There is no need to maintain the MapCount for each mapped address. Also, following default OpenMP semantics for data copying (without the *always* modifier on the map clause), data is copied to GPU memory only when it is first allocated and copied back only when it is de-allocated. As a result, there is no need to recursively traverse the data structure multiple times. Only one traversal at the beginning and one at the end of the lifetime of the mapped data is needed. Thus, there is significant potential for improving runtime performance.

**User Specified De-allocation** The runtime maintains a MapCount per address so that it can automatically determine the end of the lifetime of a mapped data item, i.e. when the data item should be copied back and de-allocated from the GPU. If the programmer is solely responsible for specifying this, e.g. by using the OpenMP *delete* map-type, then there is no need to maintain MapCounts, or to recursively traverse data structures multiple times. Thus, performance can be improved.

**Asynchronous Transfers** By default, our implementation uses synchronous data transfer calls. However, NVIDIA GPUs support asynchronous data transfers using the CUDA

[2] This excludes the use of OpenMP directives such as target enter data and target exit data.

Streams API. If sufficient bandwidth is available, multiple transfers can be overlapped for better performance. For the techniques described in Section 3.1, explicit synchronization is needed in the BASE versions when transferring data to the GPU, between the single transfer of the entire data and the subsequent transfers for fixing individual pointer values. All other transfers corresponding to the same OpenMP *map* clause can proceed in parallel.

**Selective Pointer Traversal** Prior work[5] based on OpenACC described ways for the programmer to specify which fields of a data structure to treat as pointers to be traversed in an automatic deep copy implementation. Selective pointer traversal can be applied in combination with any of the optimization techniques discussed in this section.

## 4 Experiments

In this section, we report the results of experiments performed to measure the overheads of our automatic deep copy implementation. We focused our measurements on the time taken by the runtime library calls invoked for data mapping, and on the time taken by data transfers. We ran our experiments on a system with an IBM POWER8 LE host running Linux Ubuntu 14.04, connected to an NVIDIA Kepler K40 GPU via PCIe, using CUDA version 8.0.

Our compiler system uses the IBM XL Fortran front-end to parse the OpenMP source code. It then translates the output of the front-end to Clang AST format. This Clang AST code is processed by the open-source Clang OpenMP compiler to generate a binary that executes across the host and GPU. We implement our runtime techniques by modifying the open-source runtime library that is included with the Clang OpenMP compiler. For self-managed GPU memory, we use a single call to cudaMalloc to initially allocate 2GB of GPU memory, and then manage this space in the runtime code.

We use the following benchmark codes for our evaluation:

- **List**: This code constructs and initializes a linked list of length 1024 on the host, and then traverses the list on the GPU. The type of each list element is as shown in Figure 1. There are 3 versions of the code obtained by varying the size of the list element: 128 bytes, 1KB, and 1MB.
- **SplitList**: This code uses a linked list where each list element has 2 data fields that are separated by a pointer field in the middle. As before, there are 3 versions of the code, corresponding to sizes 128 bytes, 1KB, and 1MB.
- **Tree**: This is a height-balanced binary tree with 1024 nodes. Each node has a left-child pointer, followed by a data field, followed by a right-child pointer. There are 3 versions of the code, corresponding to node sizes 128 bytes, 1KB, and 1MB.
- **UMT**: This is the kernel version of the UMT application[2], which performs three-dimensional, non-linear, radiation transport calculations. It is representative of real application code written using pointer-based data structures, and requires automatic deep copy support for easily porting it to systems with multiple memories. The data structure includes 3-level pointer chains, with multiple pointer fields at levels 2 and 3. We insert OpenMP directives to transfer 2000 nodes in the data structure to GPU memory. Total data size transferred is approximately 2.2GB.

**Results for List, SplitList, and Tree**

For benchmarks List, SplitList, and Tree, Figure 7 shows the time in seconds taken to process data transfers in the runtime. Data is separately presented for transfers to the GPU (Figure 7 (a), (b), and (c)) and transfers from the GPU (Figure 7 (d), (e), and (f)). There are 3 sizes for each benchmark, and 4 versions for each size corresponding to the different techniques described in Section 3.1.

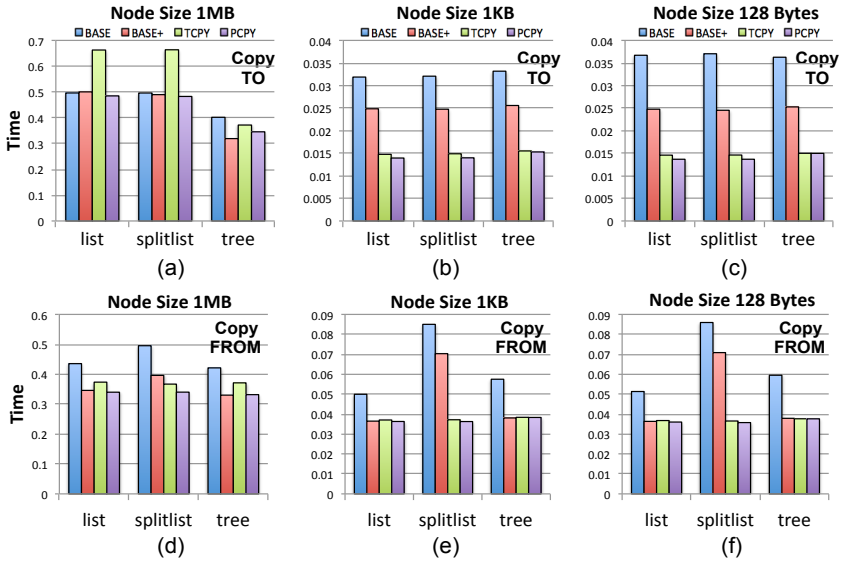


Fig. 7: Time Taken for Data Transfers (seconds)

The data in Figure 7 is used to compare the relative performance of the different versions. The low performance of the BASE version clearly shows the benefit of using self-managed memory. Overall, the results are as expected: the overhead of the extra host copy in TCPY dominates when data size is 1MB, and the overhead of extra transfers when copying to the GPU in BASE+ dominates at smaller data sizes. The results for size 128 bytes closely match those for size 1KB, as latency costs dominate the transfer time for small data sizes. Note that for SplitList, when copying from the GPU, BASE+ always has higher overhead than TCPY and PCPY. This is because SplitList has 2 data fields per node that are separately copied back to the host in BASE+.

In Figure 7(a), the versions for Tree take noticeably less time than List or SplitList. Tree has 2 pointers per node but the total number of data transfers for fixing pointer values in GPU copies in BASE+ is the same as the number of transfers for List and SplitList. This is because our runtime does not initiate any transfers for fixing pointers that are null, and the pointers in the leaf nodes of Tree are all null. Since the overall number and sizes of transfers initiated for all 3 benchmarks are similar for corresponding versions, the disparate times for Tree are due to differences in data structure traversal and clustering/sequencing of the data transfers. Profiling using nvprof shows that in this case the difference can be attributed to time spent in various CUDA API calls, while the actual transfer times are almost the same. Note that even though Tree BASE+ uses

more data transfers than Tree PCPY, it performs better for size 1MB because PCPY has greater overhead for copying the multiple pointers per node in Tree.

Figure 8 shows the percentage of effectively available bandwidth achieved for each of the testcases in Figure 7(a), (b), and (c). The effectively available bandwidth is the maximum achievable bandwidth for the pattern of transfers dictated by the data structure traversal (not the maximum bandwidth provided in hardware). We compute the effectively available bandwidth by running a manually coded CUDA version that only does GPU memory allocation/de-allocation and the sequence of data transfers corresponding to each optimization version. The CUDA version gives an optimistic upper bound on bandwidth, and it does not include any overheads of our runtime such as data structure traversal, address/offset computation, or checks and updates related to OpenMP implementation. Bandwidth is computed as the ratio of actual data transferred over the wall clock time taken to execute the code that processes transfers. The percent bandwidth achieved compared to the optimistic CUDA version is a measure of the overhead in the OpenMP runtime library code. Note that this overhead depends on data size for some optimization cases.

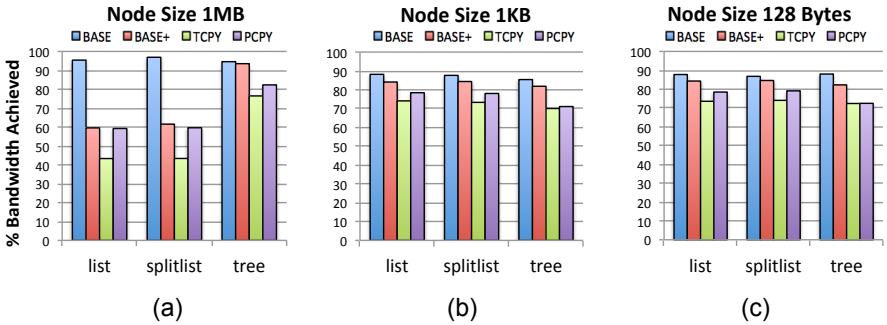


Fig. 8: Percentage Bandwidth Achieved Compared to Optimistic CUDA Version

In all cases except List and SplitList for 1MB, we achieve 70% or greater of the optimistic maximum bandwidth. As expected, the absolute values of the bandwidth are proportional to the data size, e.g. bandwidth values for size 1MB are an order of magnitude larger than the values for size 1KB. Also, for a given benchmark/size, the optimistic bound computed for BASE is lower than that computed for BASE+, which in turn is lower than that computed for TCPY and PCPY. This explains why the percent bandwidths achieved by BASE and BASE+ are relatively higher even though they spend more time processing data transfers. On average across all cases, 77.5% of the effectively available bandwidth is achieved.

We also implemented a version of our runtime using asynchronous data transfers with two CUDA streams. However, for our testcases, the overheads associated with asynchronous transfers (allocating/copying to pinned host memory, and API calls for synchronization) caused slowdowns in overall performance. Further experiments are needed to determine if these overheads can be overcome.

**Results for UMT**

We also measured the performance of automatic deep copy for transferring data to the GPU in the UMT benchmark. For each version BASE, BASE+, TCPY, and PCPY, Table 2 shows the time in seconds to process the data transfer, and the bandwidth of the

transfer in GB/s. As a reference, the absolute values of the bandwidths achieved by the 1MB size testcases in Figure 8(a) ranged from 1.615GB/s to 3.358GB/s. The results of our initial experiments indicate that the overhead of automatic deep copy may be tolerable for practical use cases.

Table 2: UMT Transfers to GPU Memory

| | **BASE** | **BASE+** | **TCPY** | **PCPY** |
|---|---|---|---|---|
| Time (seconds) | 5.4382 | 3.3708 | 2.6548 | 2.3492 |
| Bandwidth (GB/s) | 0.4367 | 0.7045 | 0.8944 | 1.0107 |

## 5 Related Work

Prior work related to OpenACC[5, 17] has addressed the issue of designing automatic deep copy traversals, and it is supported to some extent in the Cray and PGI Fortran compilers. However, overheads associated with deep copy are not well understood. In our work, we described and implemented a specific algorithm for deep copy that also supports cyclic pointer traversals, proposed optimization techniques based on this algorithm, and performed experiments to measure the overheads of different techniques.

The main advantage of our automatic deep copying approach is it enables ease of programming. Software shared memory abstractions (e.g. [4, 14, 11]) provide another way to make programming easier. CUDA Unified Memory(UM)[9] is a shared memory abstraction available on systems with NVIDIA GPUs. UM is an on-demand solution that works on OS page-size granularity, and can have very high overhead in some cases. In contrast, our approach can incorporate prefetching optimizations, and can be specifically optimized for each application's data structures and access patterns.

The system used in our experiments has a PCIe interconnect between the CPU and GPU. NVLink[7] is a custom high-bandwidth interconnect that can be used with NVIDIA GPUs. We expect that using a system with NVLink will help reduce the overheads associated with automatic deep copies.

Our implementation is based on OpenMP. The directives for data mapping in OpenACC are very similar to those in OpenMP. There are other high-level paradigms for programming heterogeneous systems, such as C++ AMP[8] and Kokkos[6], both of which use the concept of data views. These aim to enable performance portability for data accesses; they do not provide support for automatically traversing recursive pointer-based data structures.

Garbage collection[13] techniques for memory management automatically track the lifetimes of pointer-based data. In our algorithm, we also track the lifetime of data encountered in deep copy traversals, except our case is simpler because we follow OpenMP semantics. Specifically, we only track the number of variables directly specified in map clauses that may reach a given data item through deep copy traversal.

In our work, we rely on Fortran language features to completely automate deep copy traversals. For other languages such as Java/C/C++, there exist libraries and APIs for serialization that can be used to partially automate deep copy traversals.

## 6 Conclusion

We designed and implemented automatic support for deep copy of pointer-based data structures across multiple memories. We proposed several techniques that can be applied to optimize the overhead of pointer-based data transfers. We obtained experimental data to evaluate the overheads of our implementation in a CPU-GPU system, and to determine the applicability of the different techniques proposed. Overall, our work shows that automatic copying of pointer-based data structures can be implemented using the compiler and runtime with manageable overheads.

## Acknowledgement

This work was supported in part by the United States Department of Energy CORAL program (contract B604142).

## References

1. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.
2. CORAL Benchmark Codes: Single node UMT microkernel. `https://asc.llnl.gov/CORAL-benchmarks/\#umtmk`, 2014.
3. OpenMP Application Programming Interface, v4.5. `http://openmp.org/wp/openmp-specifications`, 2015.
4. B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. *Compcon Digest of Papers.*, 1993.
5. J. Beyer, D. Oehmke, and J. Sandoval. Transferring user-defined types in OpenACC. *Proceedings of Cray User Group*, 2014.
6. H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos. *Journal of Parallel and Distributed Computing*, 74(12), 2014.
7. D. Foley. NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data. `https://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data`.
8. K. Gregory and A. Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*. Microsoft Press, 2012.
9. M. Harris. Unified Memory in CUDA 6. `https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6`.
10. HSA Foundation. HSA Runtime Programmer's Reference Manual, version 1.1, 2016.
11. L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory of Computing Systems*, 31(4), 1998.
12. T. Jablin, J. Jablin, P. Prabhu, F. Liu, and D. August. Dynamically Managed Data for CPU-GPU Architectures. *International Symposium on Code Generation and Optimization*, 2012.
13. R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.
14. P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *International Symposium on Computer Architecture (ISCA)*, 1992.
15. NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010.
16. NVIDIA Corporation. PGI Accelerator Compilers OpenACC Getting Started Guide, 2016.
17. OpenACC-Standard.org. The OpenACC application programming interface, v2.5, 2015.
18. C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. *Programming Language Design and Implementation*, 2010.