Transferring User Defined Types in OpenACC

James Beyer, David Oehmke, Jeff Sandoval
(Cray Inc.)

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

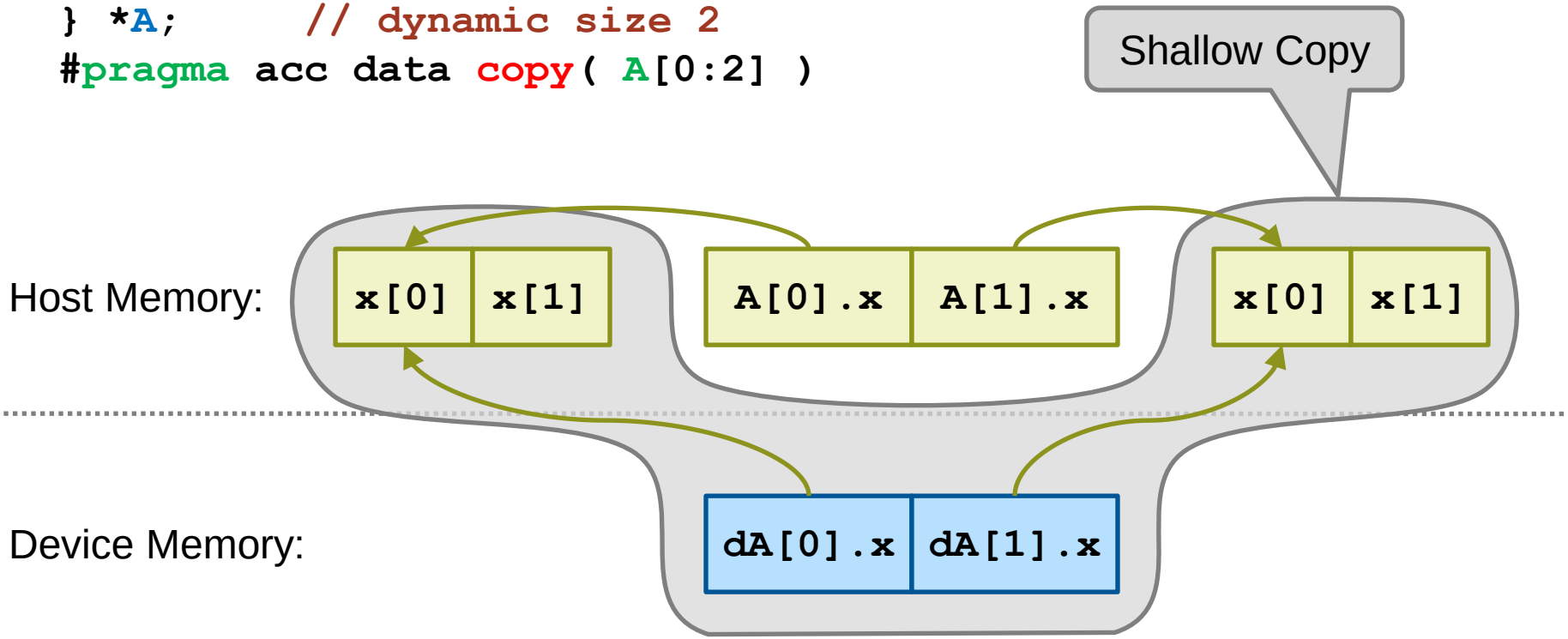
Topics

- What is this “transferring user define types thing”
- Existing solutions
- Deep copy capabilities
- Directive based solutions
- Complications and future work

Disjoint data-structure challenges

- Non-contiguous transfers
- Pointer translation

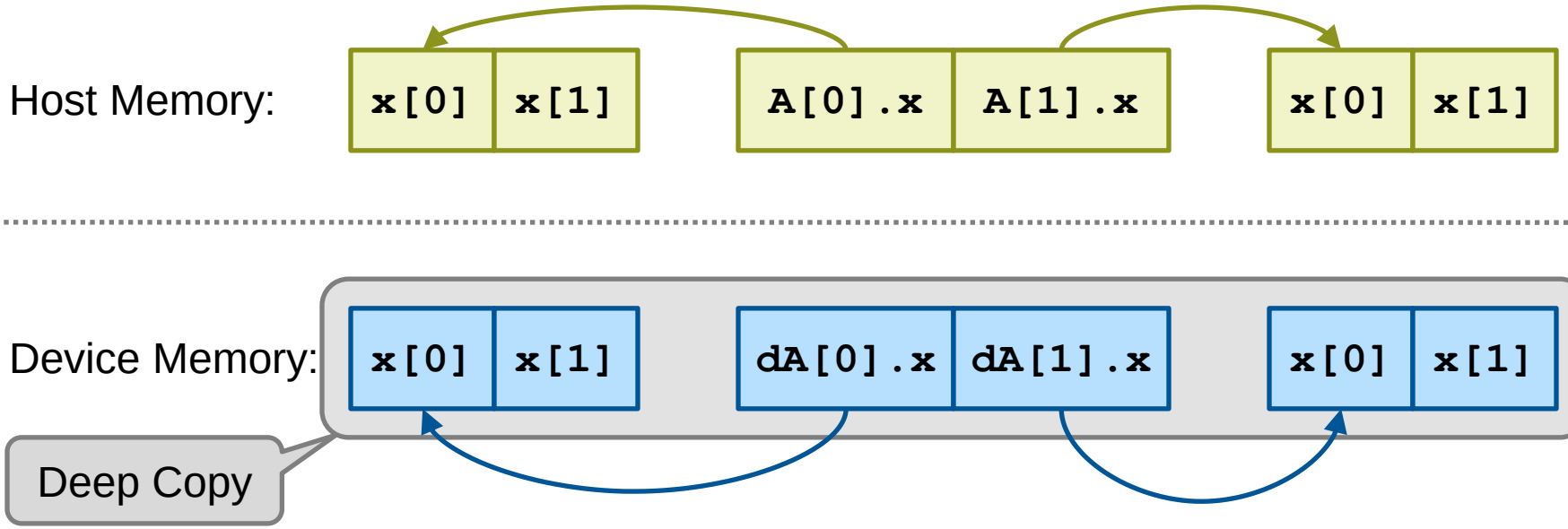
```
struct {
  int *x; // dynamic size 2
} *A;     // dynamic size 2
#pragma acc data copy( A[0:2] )
```



Disjoint data-structure challenges

- Non-contiguous transfers
- Pointer translation

```
struct {
  int *x; // dynamic size 2
} *A;     // dynamic size 2
#pragma acc data copy( A[0:2] )
```



Transferring user defined types

- **MPI**
  - MPI_Type_contiguous()
  - MPI_Type_vector()
  - MPI_Type_indexed()
- **Object serialization**
  - Write structures to storage and reload later
  - Supported in many languages
- **OpenACC**
  - API
  - Directives
- **“Deep copy”**

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

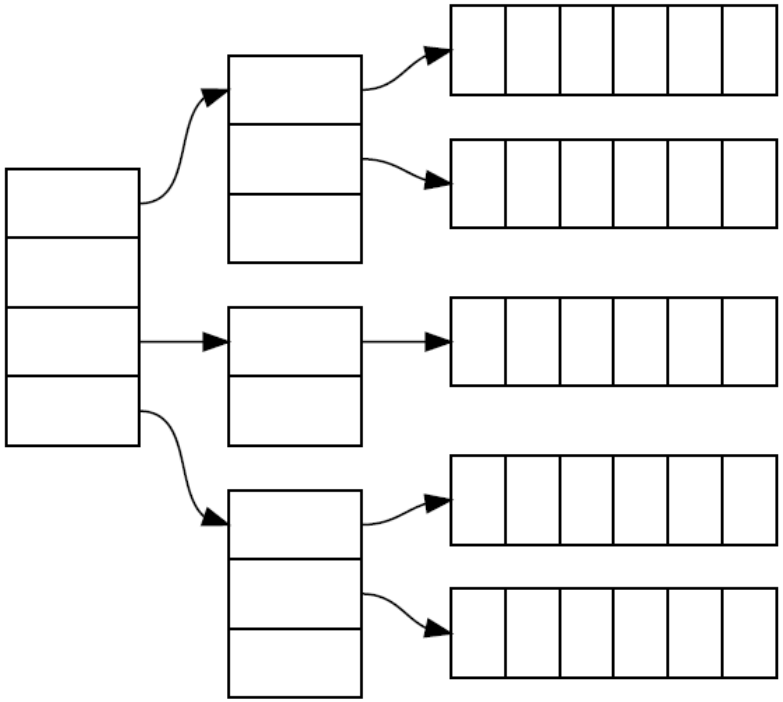
Manual deep copy example

```
void deep_copy(  struct Deep* P, int n ) {
  int i,j;
  struct Deep *dP;
  double *dA, *dB;
/* enter copyin( P[0:n] ) */
  dP = acc_copyin( P, sizeof( struct Deep)*n );
  for ( i=0; i < n; ++i ) {
    dA = acc_copyin( P[i].A, P[i].size*sizeof( double ) );
    acc_memcpy_to_device( &dP[i].A, &dA, sizeof( double* ) );
    dB = acc_copyin(P[i].B, P[i].size*sizeof( double ));
    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) {
    acc_update_self( &P[i].scalar, sizeof(double) );
    acc_copyout( P[i].A, P[i].size* sizeof(double) );
    acc_copyout( P[i].B, P[i].size* sizeof(double) );
  }
  acc_delete( P, sizeof( struct Deep )*n );
}
```

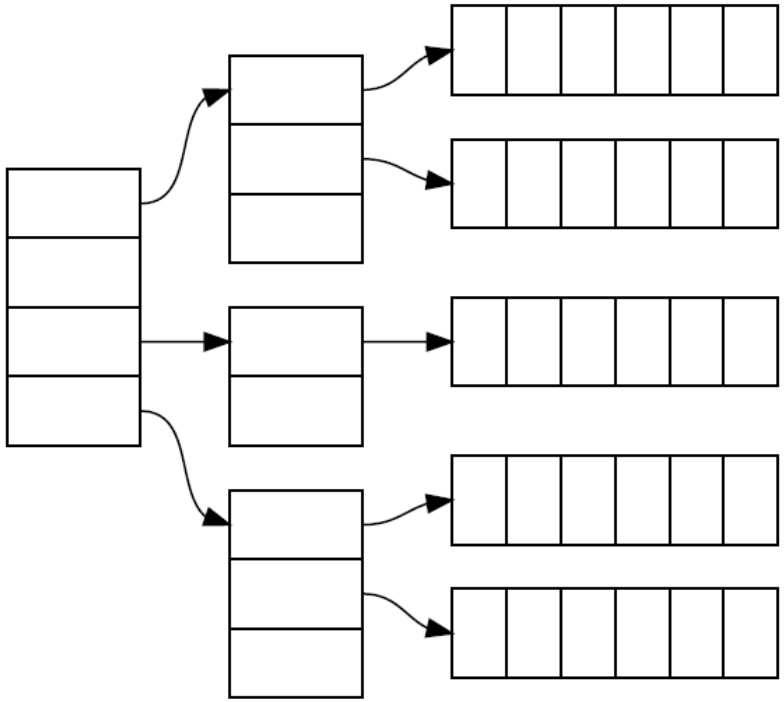
Language complications

- **Fortran allocatable and pointer members**
  - Self-describing
  - Compiler can calculate their shape at runtime
  - Opaque types
    - Difficult to use manual deep copy, pointer hidden
- **C/C++ pointers**
  - Compiler has no way to calculate their shape
  - User can easily manipulate since it's a basic type
- **Solution**
  - Allow user to provide shaping information for C/C++ pointer members
  - Directives allow compiler to manage dope vectors during transfer

Full deep-copy

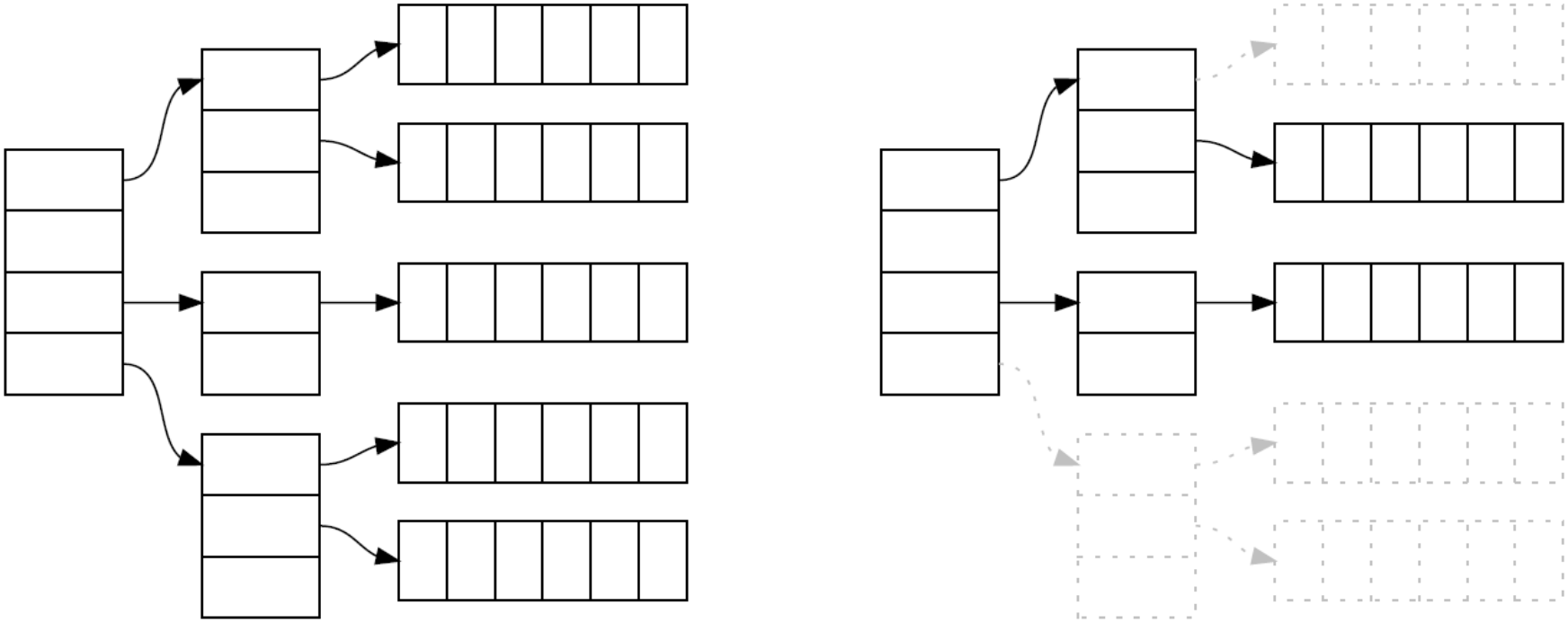


(a) Host memory



(b) Device memory

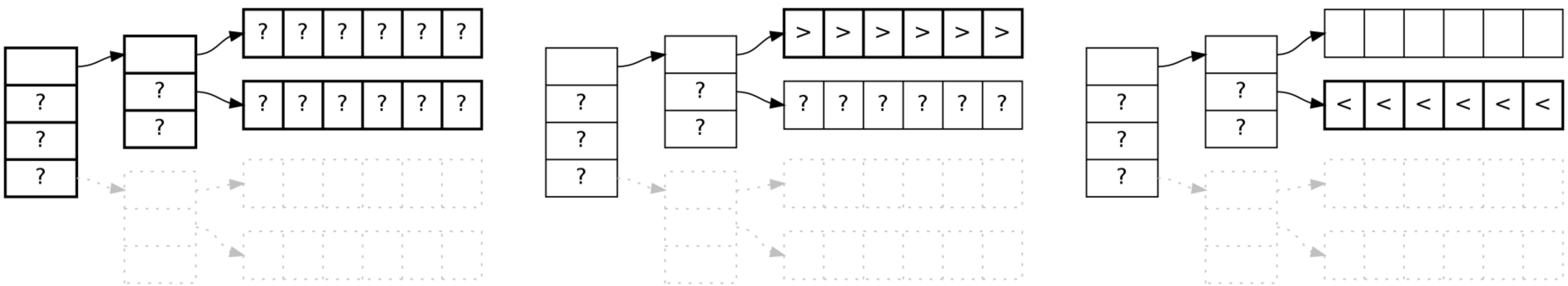
Selective member deep-copy



(a) Host memory

(b) Device memory

Selective direction deep-copy

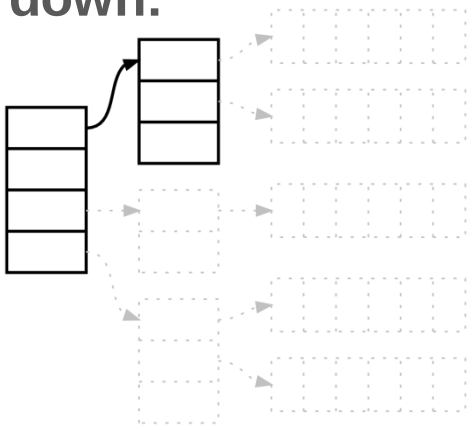


(a) Deep `create` (b) Selective `copyin` (c) Selective `copyout`

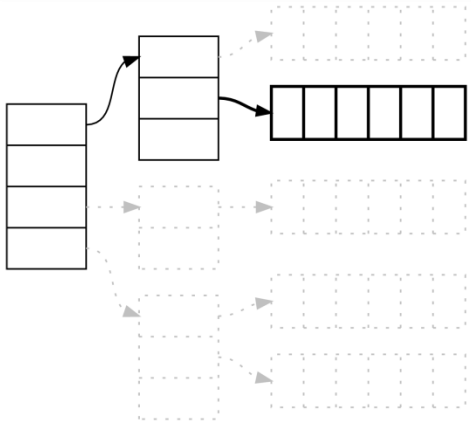
- **This is an optimization**
  - Users can transfer everything in and out
- **This is a convenience**
  - Users can do this with deep create and selective update

Mutable deep-copy

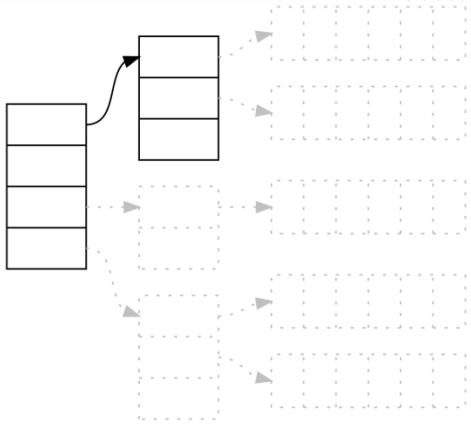
Top down:



(a) Selective deep-copy

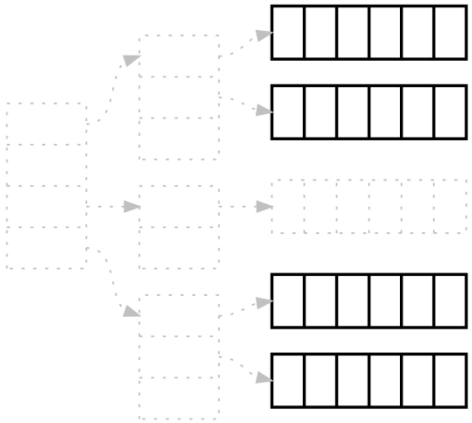


(b) Attach to parent

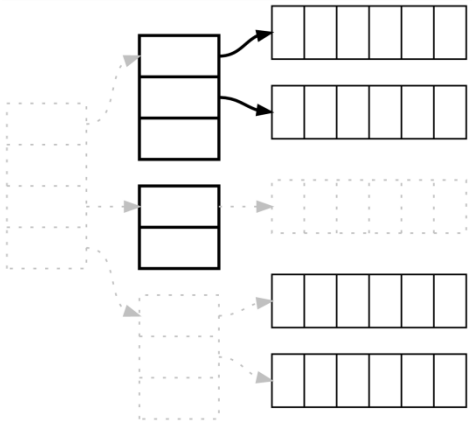


(c) Detach from parent

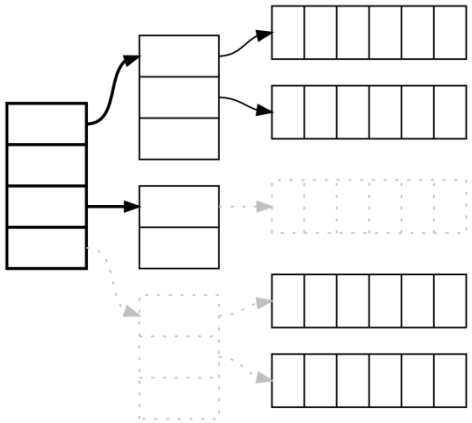
Bottom up:



(a) Copy sub-objects



(b) Attach to children



(c) Attach to children

Local Directive

```
struct Deep {
int size;
double scalar;
double* A; /* A[0:size] */
double* B; /* B[0:size] */
}*p;
// copy n elements of p and size elements of A and B for each p
#pragma acc data copy( p[0:n]::{ copy( A[0:size], B[0:size] ) }
// copyin p and B, copyout A
#pragma acc data copyin( p[0:n])::{copyout( A[0:size] ),
                                   copyin ( B[0:size] )}
// copy p, copy A, leave B as shallow copy
#pragma acc data copy( p[0:n] )::{ copy( A[0:size] ) }
// verify presence of p, copy and attach B, leave A unchanged
#pragma acc data present( p[0:n] )::{ copy( B[0:size] )}
```

Global Policies

```
struct Deep {
  … // same as last slide
#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[*] )
};
// Use deep policy to copy p, copy A and B
#pragma acc data copy( deep : p[0:n] )
// Use sel_dir policy to copyin p, copyout A and copyin B
#pragma acc data copyin( sel_dir : p[0:n] )
// Use sel_mem policy to copyin p, copy A
#pragma acc data copyin( sel_mem : p[0:n] )
// Use mut_copy policy to attach B to existing p
#pragma acc data present( mut_copy : p[0:n] )
```

Implicit policies, template

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

…
// Shape _begin to size of active elements, others are aliases
#pragma acc policy( shape )                                          \
    shape( _begin[0:(((_end_data - _begin)/sizeof(T)) - 1)] )
// Create implicit data policy using _begin shape policy
#pragma acc policy( data ) inherit( _begin[*] )                      \
                           present( _end_data[@_begin] )             \
                           present( _end_storage[@_begin] )
// Create implicit update policy using _begin shape policy
#pragma acc policy(update) update( _begin[*] )                       \
                           maintain( _end_data, _end_storage )
};
```

Implicit policies, use template

```
class Data {
private:
    my_vector<double> A;
    my_vector<double> B;
    my_vector<int>    C;
    my_vector<int>    Other;
// Crate shape policy using shape from template
// for A, B and C, other is forced shallow
#pragma acc policy( shape )  shallow( Other )
// Create data policy using template data policy and shape
#pragma acc policy( data )   inherit(*)
// Create update policy using template update policy
#pragma acc policy( update ) update(*)
};
```

Implicit policies, use template and class

```
// Example 1

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 )

//Example 2

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) }
```

Complications and future work

- Modifying pointers in data regions
- Allocation/deallocation on the device
- Polymorphic objects
- Function pointers
- C++ “this->” shallow or deep?
- C++ templates
- etc.

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.