

EXPERIENCES IN IMPLEMENTING OPENMP OFFLOAD SUPPORT IN FORTRAN

Kostas Makrides – HPE

Aaron Black – LLNL

COPYRIGHT AND TRADEMARK ACKNOWLEDGEMENTS

©2016-2021 Cray, a Hewlett Packard Enterprise company. All Rights Reserved.

Portions Copyright Advanced Micro Devices, Inc. (“AMD”) Confidential and Proprietary.

The following are trademarks of Cray, and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray: APPRENTICE2, CHAPEL, CLUSTER CONNECT, CLUSTERSTOR, CRAYDOC, CRAYPAT, CRAYPORT, DATAWARP, ECOPHLEX, LIBSCI, NODEKARE, and REVEAL. The following system family marks, and associated model number marks, are trademarks of Cray: CS, CX, XC, XE, XK, XMT, and XT. ARM is a registered trademark of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. ThunderX, ThunderX2, and ThunderX3 are trademarks or registered trademarks of Cavium Inc. in the U.S. and other countries. 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. Intel, the Intel logo, Intel Cilk, Intel True Scale Fabric, Intel VTune, Xeon, and Intel Xeon Phi are trademarks or registered trademarks of Intel Corporation in the U.S. and/or other countries. Lustre is a trademark of Xyratex. NVIDIA, Kepler, and CUDA are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and/or other countries.

Other trademarks used in this document are the property of their respective owners.



AGENDA

- Introduction and important considerations when implementing OpenMP offload support in HPC Fortran Applications
- Memory allocators and Interoperability
 - Using the OpenMP API to create interfaces for memory allocations outside of OpenMP
- Mapping non-trivial data structures to the device
 - Pitfalls in moving pointer components to the device
- Review with a final example
 - Exposing non-intuitive behaviors from the OpenMP runtime.
- We will use examples throughout the talk from a repo from Github that contains many OpenMP use-cases in HPC production codes <https://github.com/LLNL/FGPU>



MAJOR CONSIDERATIONS FOR HPC APPLICATIONS

- Explicit control over data movements
 - This usually requires the usage of standalone directives
 - Managing communication and data movements correctly can have benefits to performance
- Improve quality-of-life management of code base
 - Code Readability
 - Maximize code reuse , avoid creating many relatively similar subroutines
 - Support debugging solutions
- Interoperability with other HPC applications and tools
 - Most applications aren't completely sandboxed and isolated
 - Technologies have been created to help manage painful tasks such as memory management
 - For example: <https://github.com/LLNL/Umpire>
- OpenMP is a viable and reliable solution for Fortran applications running on accelerators
 - Performance and portability are attractive features

EXPLICIT CONTROL OVER DATA MOVEMENTS

call init(...)

!\$omp target map(tofrom: X, Y)

!\$omp end target

call init(...)

!\$omp target enter data map(to: X)

!\$omp target enter data map(to: Y)

!\$omp target

!\$omp end target

!\$omp target exit data map(from: Y)

!\$omp target exit data map(from: X)



QUALITY OF LIFE

```
subroutine map_dt1(X(:))  
!$omp target map(to: X)
```

```
end subroutine
```

```
subroutine map_dt2(X(:, :))  
!$omp target map(to: X)
```

```
end subroutine
```

```
subroutine map_dtN(Z(:, :))  
!$omp target map(to: X)
```

```
end subroutine
```



```
subroutine critical_subroutine(...)
```

```
!$omp target enter data map(to:foo_data(offset)%d1x)  
!$omp target enter data map(to:foo_data(offset)%d1y)  
!$omp target enter data map(to:foo_data(offset)%d1z)  
!$omp target enter data map(to:foo_data(offset)%d2x)  
!$omp target enter data map(to:foo_data(offset)%d2y)  
!$omp target enter data map(to:foo_data(offset)%d2z)
```

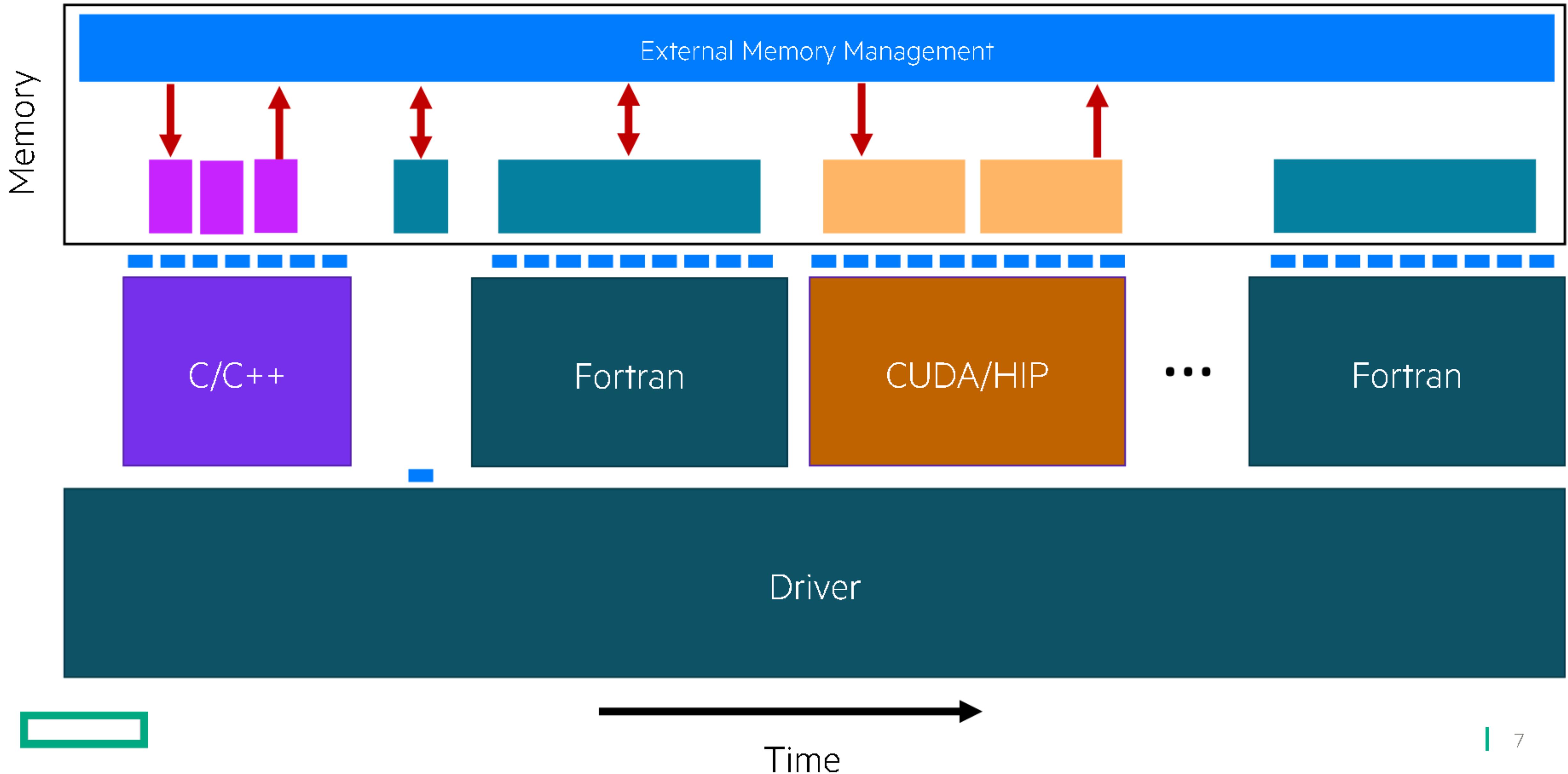
```
call map_dt1(foo_data(offset)%dt1x)
```

```
!$omp target
```

```
!$omp end target
```

```
!$omp target exit data map(to:foo_data(offset)%d1x)  
!$omp target exit data map(to:foo_data(offset)%d1y)  
!$omp target exit data map(to:foo_data(offset)%d1z)  
!$omp target exit data map(to:foo_data(offset)%d2x)  
!$omp target exit data map(to:foo_data(offset)%d2y)  
!$omp target exit data map(to:foo_data(offset)%d2z)
```

HPC ENVIRONMENT INTEROPERABILITY



MEMORY ALLOCATORS AND INTEROPERABILITY

- Modern HPC applications often have a mix of different language and programming models
 - HIP/CUDA
 - OpenMP C/C++
- The management of memory from these different parts of the code can be unified through a shared memory management library (For example: Umpire <https://github.com/LLNL/Umpire>)
 - This can be critical for performance
- OpenMP has low level hooks allowing you to use device memory allocated outside of OpenMP
 - Users able to manually associate host and device memory addresses.
- Many of these OpenMP API functions regarding memory were previously only available in C/C++ and required the user to generate Fortran interfaces
 - As of OpenMP 5.1, this is no longer the case (at least for many of the API functions)



BASIC EXAMPLE OF USING AN EXTERNAL ALLOCATOR

OpenMP API available was limited to C interface

```
subroutine testsaxpy_omp45_f
use iso_c_binding
use omp_lib
implicit none
integer, parameter :: N = ishft(1,21)
integer :: i, err
real, pointer :: x(:), y(:)
type(c_ptr) :: x_cptr, y_cptr
integer(c_size_t) :: num_bytes = sizeof(a)*N, offset=0
real :: a = 2.0
allocate(x(N), y(N))
x = 1.0
y = 2.0

x_cptr = omp_target_alloc(num_bytes, omp_get_default_device())
err = omp_target_associate_ptr(C_LOC(x), x_cptr, num_bytes, offset, omp_get_default_device())

y_cptr = omp_target_alloc(num_bytes, omp_get_default_device())
err = omp_target_associate_ptr(C_LOC(y), y_cptr, num_bytes, offset, omp_get_default_device())

!$omp target update to(x,y)
!$omp target data map(to:N,a)

!$omp target teams distribute parallel do private(i) shared(y,a,x) default(none)
do i=1,N
    y(i) = a*x(i) + y(i)
end do
!$omp end target teams distribute parallel do
!$omp end target data

!$omp target update from(y)
err = omp_target_disassociate_ptr(C_LOC(x), omp_get_default_device())
call omp_target_free(x_cptr, omp_get_default_device())
err = omp_target_disassociate_ptr(C_LOC(y), omp_get_default_device())
call omp_target_free(y_cptr, omp_get_default_device())
write(*,*) "Ran FORTRAN OMP45 kernel. Max error: ", maxval(abs(y-4.0))
end subroutine testsaxpy_omp45_f
```

Allocate a block of memory on the device

Associate it with a host pointer
Note: Only the buffer address is now mapped, not the dope vector.

Relying on runtime to implicitly map dope vectors.

Dissociate the host pointer with the device memory

Free the block on the device

BASIC EXAMPLE OF USING AN EXTERNAL ALLOCATOR

Runtime behavior was as expected.

```
subroutine testsaxpy_omp45_f
use iso_c_binding
use omp_lib
implicit none
integer, parameter :: N = ishft(1,21)
integer :: i, err
real, pointer :: x(:), y(:)
type(c_ptr) :: x_cptr, y_cptr
integer(c_size_t) :: num_bytes = sizeof(a)*N, offset=0
real :: a = 2.0
allocate(x(N), y(N))
x = 1.0
y = 2.0

x_cptr = omp_target_alloc(num_bytes, omp_get_default_device())
err = omp_target_associate_ptr(C_LOC(x), x_cptr, num_bytes, offset, omp_get_default_device())

y_cptr = omp_target_alloc(num_bytes, omp_get_default_device())
err = omp_target_associate_ptr(C_LOC(y), y_cptr, num_bytes, offset, omp_get_default_device())

 !$omp target update to(x,y)
 !$omp target enter
 ACC: Simple transfer of 'x(:)' (72 bytes)
 ACC: host ptr 7fffffff9010
 ACC: acc_ptr 0
 ACC: flags: DOPE_VECTOR DV_ONLY_DATA COPY_HOST_TO_ACC REG_PRESENT
 INIT_ACC_PTR IGNORE_ABSENT
 !$omp end target
 !$omp end target
 !$omp target update from(x,y)
 err = omp_target_free(x_cptr, omp_get_default_device())
err = omp_target_disassociate_ptr(C_LOC(y), omp_get_default_device())
call omp_target_free(y_cptr, omp_get_default_device())
write(*,*) "Ran FORTRAN OMP45 kernel. Max error: ", maxval(abs(y-4.0))
end subroutine testsaxpy_omp45_f
```

```
ACC: allocate <internal> (8388608 bytes)
ACC: allocate <internal> (8388608 bytes)
ACC: Start transfer 2 items from saxpy_omp45_f.F90:88
ACC: copy to acc 'x(:)' (8388608 bytes)
ACC: copy to acc 'y(:)' (8388608 bytes)
ACC: End transfer (to acc 16777216 bytes, to host 0 bytes)
```

USING AN EXTERNAL ALLOCATOR WITH SUBROUTINE API

Extra code placed in subroutines

```
subroutine testsaxpy_omp45_f
use iso_c_binding
use omp_lib
implicit none
integer, parameter :: N = ishft(1,21)
integer :: i, err
real, pointer :: x(:), y(:)
type(c_ptr) :: x_cptr, y_cptr
integer(c_size_t) :: num_bytes = sizeof(a)*N, offset=0
real :: a = 2.0
allocate( x(N), y(N) )
x = 1.0
y = 2.0
```

! Allocation and associate pointers code in subroutine.

```
call map_array(x)
```

Allocate a block of memory on the device

!\$omp target teams distribute parallel do private(i) shared(y,a,x) default(none)

```
do i=1,N
    y(i) = a*x(i) + y(i)
end do
!$omp end target teams distribute parallel do
!$omp end target data
!$omp target update from(y)
```

!Disassociate pointers and deallocation code in subroutine.

```
call unmap_array(x)
```

Associate it with a host pointer

```
write(*,*) "Ran FORTRAN OMP45 kernel. Max error: ", maxval(abs(y-4.0))
```

```
end subroutine testsaxpy_omp45_f
```

Dissociate the host pointer with the device memory

Free the block on the device

USING AN EXTERNAL ALLOCATOR WITH SUBROUTINE API

- Hiding the repetitive code within the subroutine calls creates simplicity and makes things less error prone
 - Quality-of-life improvements
- If you're mapping simple things (arrays) you don't have many map subroutine flavors that you need to manage
 - For example: integers, floats,
 - Even a few derived types* shouldn't have a lot of overhead
- What would happen if you passed a pointer component of a derived type?
 - call map_array(foo%foo_arr)

```
subroutine map_array(x,N)
use iso_c_binding
use omp_lib
implicit none
integer :: N, num_bytes=N*4
real :: x(:)
type(c_ptr) :: x_cptr

allocate( x(N) )
x_cptr = omp_target_alloc(num_bytes, omp_get_default_device() )
err = omp_target_associate_ptr( C_LOC(x), x_cptr, num_bytes, 0,
omp_get_default_device() )
if (err /= 0) then
    print *, "Target associate on x failed."
endif
!$omp target update to(x)

end subroutine map_array
```



ADDING DERIVED TYPES TO THE MIX

- Nested data is often a stress point in implementing OpenMP offload in Fortran
- Runtime behavior differs in Fortran vs C.

“Each pointer component that is a list item that results from a mapped derived type variable is treated as if its association status is undefined, unless the pointer component appears as another list item or as the base pointer of another list item in a map clause on the same construct.”

- Notice the distinction with the rules for a c struct
- An explicit map of the pointer-component of the derived type is needed

```
!$omp target enter data map(to:var,var%comp_b)
```

or

```
!$omp target enter data map(to:var%comp_b)
```

Note: The OpenMP runtime will implicitly map the base derived type “var”

- This is new in the 5.0 standard and the map clause doesn't explicitly address this in the OpenMP 4.5 standard
 - Each compiler may have been treating derived-type pointer components differently
- The requirements from the standard have changed from 4.5 to 5.0/5.1
 - The 4.5 standard didn't explicitly address derived types with pointer components.



APPLICATION USE CASE

Mapping Derived Types with Pointer components

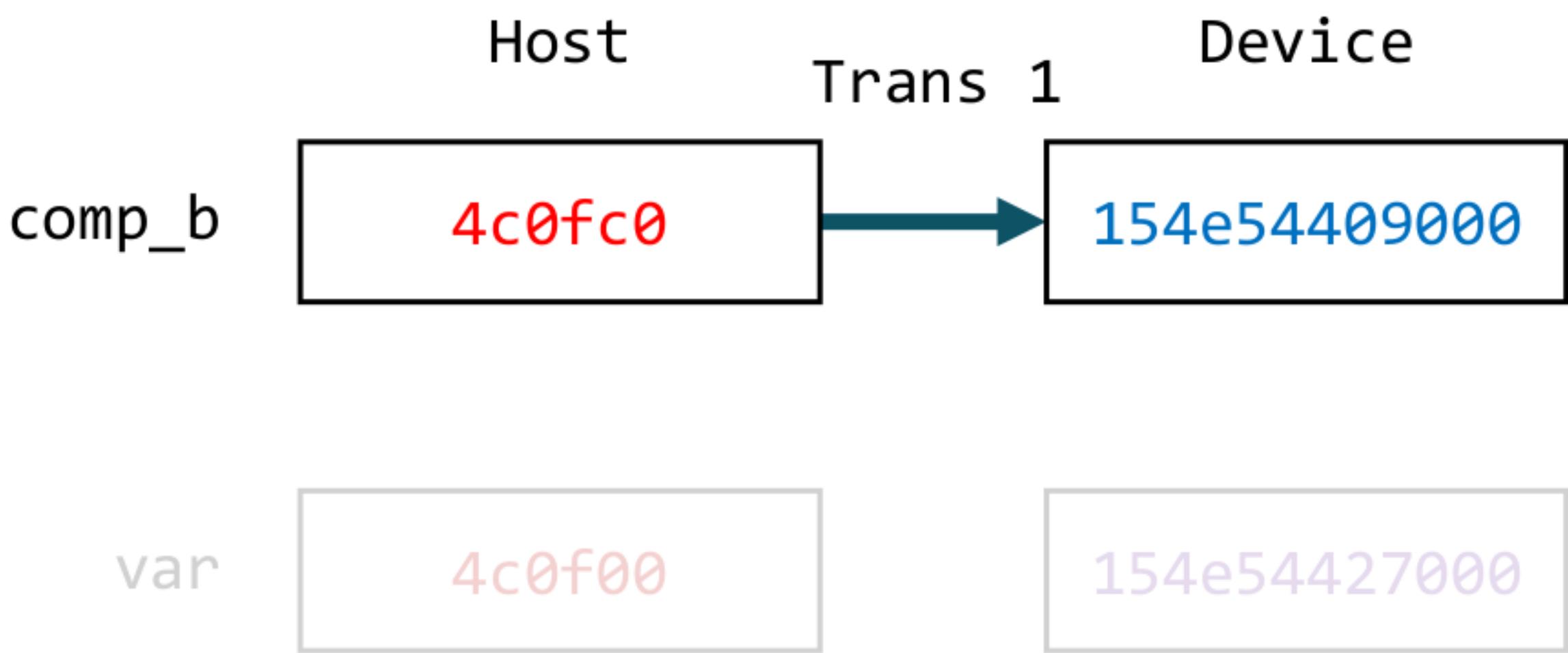
- This is a reproducer that mimics attempts to use the previously mentioned mapping subroutines on nested data.
- The key insight here was understanding what the implications of the `map_array` subroutine
 - Note: We explicitly added a directive into the `test_mapper` program which explicitly tells the compile not to inline any function or subroutine calls (as is in the real application)
- If the `map_array` subroutine is inlined then the subsequent OpenMP directive would effectively be:
`!$omp target enter data map(to:var%comp_b)`
which is functionally different to
`!$omp target enter data map(to:h_ptr)`
 - The difference is the pointer attachment on the device!
- I will use some available tools to elucidate the aforementioned point
 - Two cases will be shown, one when the map array is inlined and one when it is not.

```
1 module test_map
2   type type_a
3     integer, pointer, contiguous :: comp_b(:)
4   end type type_a
5
6 contains
7
8   subroutine map_array(h_ptr)
9     implicit none
10
11   integer, pointer :: h_ptr(:)
12
13   !$omp target enter data map(to:h_ptr)
14 end subroutine map_array
15 end module test_map
16
17 program test_mapper
18 !DIR$ NOINLINE
19 use test_map
20 implicit none
21 integer, parameter :: n=30000
22 integer :: i
23
24 type(type_a), allocatable:: var
25 allocate(var)
26 allocate(var%comp_b(n))
27
28 call map_array(var%comp_b)
29
30 !$omp target teams distribute simd
31 do i=1,n
32   var%comp_b(i) = i
33 end do
34
35 end program test_mapper
```

APPLICATION USE CASE

Mapping Derived Types with Pointer components, inlined

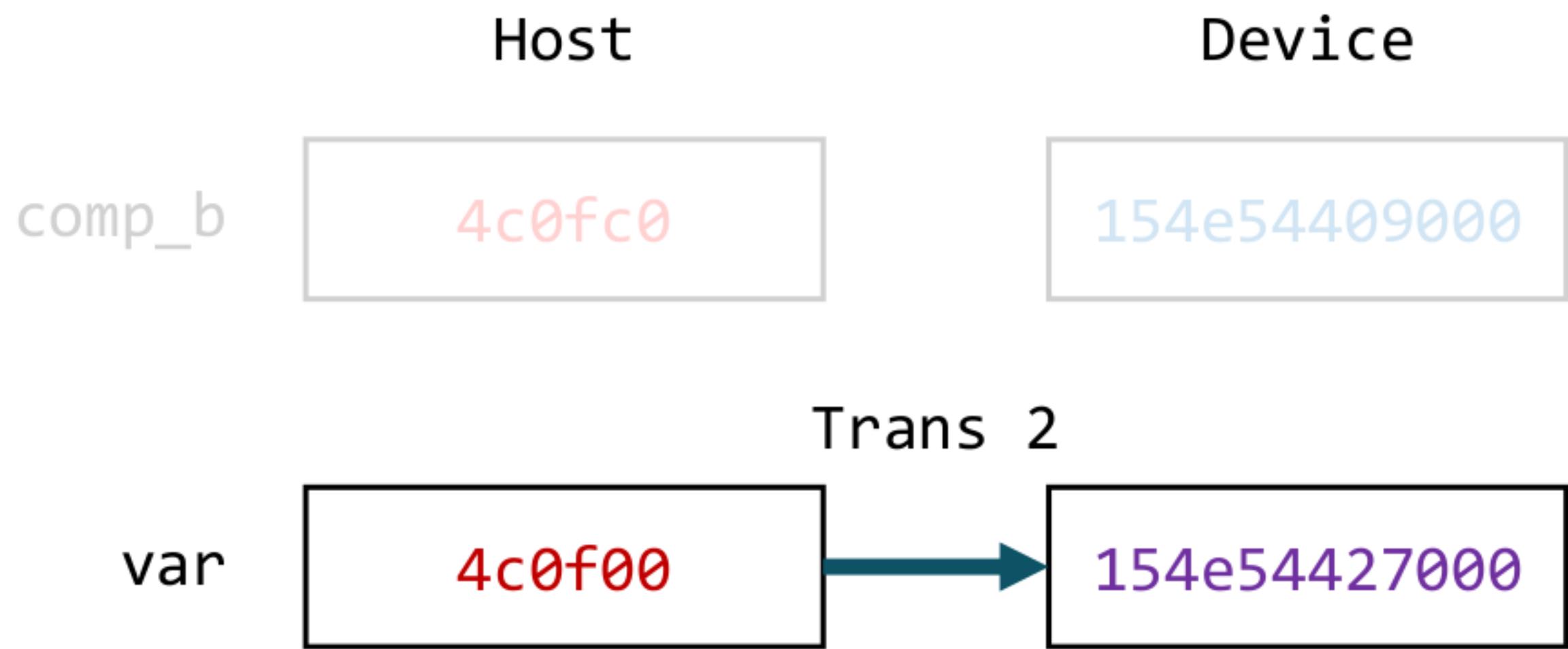
```
ACC: Start transfer 3 items from origMain.F90:28
ACC:   flags: NEED_POST_PHASE
ACC: Transfer Phase
ACC: Trans 1
ACC:     Simple transfer of 'var%comp_b(:)' (72 bytes)
ACC:       host ptr 4c0f00
ACC:       acc  ptr 0
ACC:       flags: DOPE_VECTOR DV_ONLY_DATA ALLOCATE
COPY_HOST_TO_ACC ACQ_PRESENT REG_PRESENT
ACC:     Transferring dope vector
ACC:       DV size=120000 (dim:1 extent:30000
stride_mult:1 scale:4 elem_size:4)
ACC:       total mem size=120000 (dv:0 obj:120000)
ACC:       memory not found in present table
ACC:       allocate (120000 bytes)
ACC:         get new reusable memory, added entry
ACC:       new allocated ptr (154e54409000)
ACC:       add to present table index 0: host 4c0fc0 to
4de480, acc 154e54409000
ACC:       copy host to acc (4c0fc0 to 154e54409000)
ACC:         internal copy host to acc (host 4c0fc0 to
acc 154e54409000) size = 120000
ACC:       new acc ptr 154e54409000
```



APPLICATION USE CASE

Mapping Derived Types with Pointer components, inlined

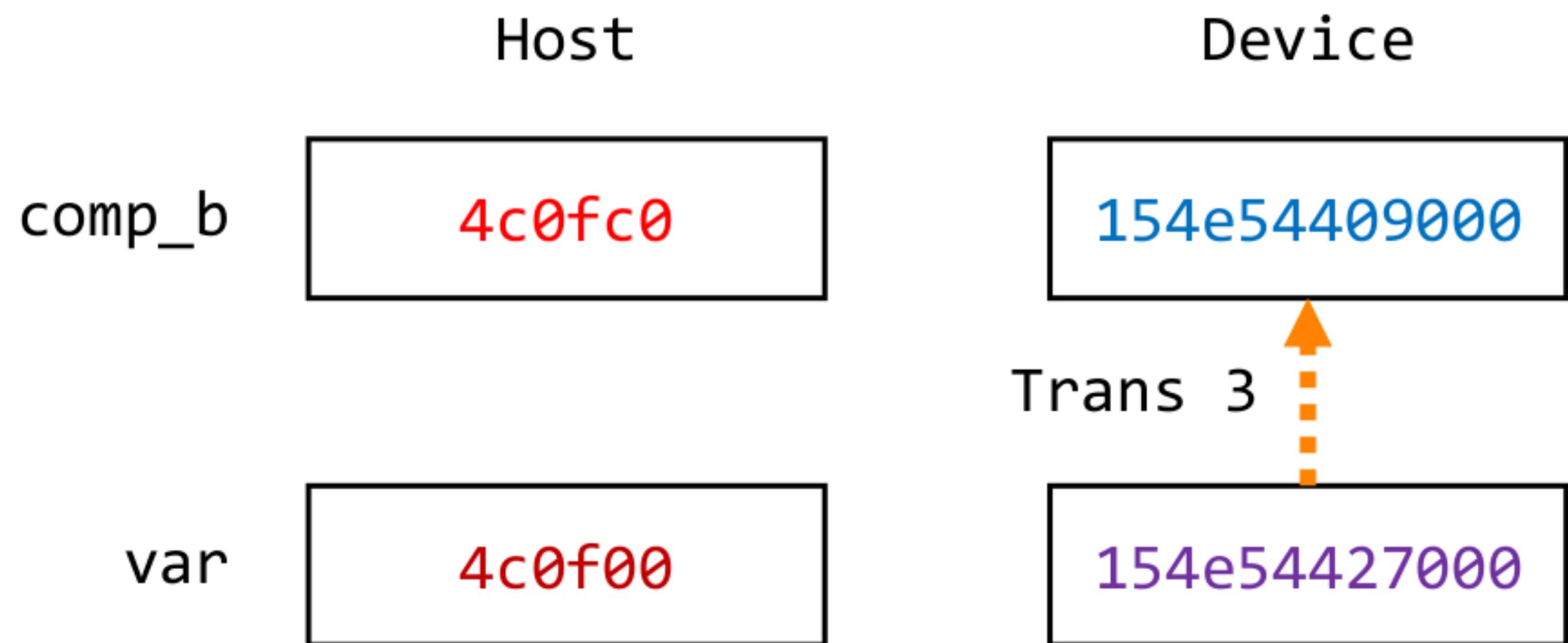
```
ACC: Trans 2
ACC:     Simple transfer of 'var' (72 bytes)
ACC:         host ptr 4c0f00
ACC:         acc  ptr 0
ACC:         flags: ALLOCATE ACQ_PRESENT
REG_PRESENT
ACC:         memory not found in present table
ACC:         allocate (72 bytes)
ACC:             get new reusable memory, added entry
ACC:             new allocated ptr (154e54427000)
ACC:             add to present table index 1: host
4c0f00 to 4c0f48, acc 154e54427000
ACC:             new acc  ptr 154e54427000
```



APPLICATION USE CASE

Mapping Derived Types with Pointer components, inlined

```
ACC: Trans 3
ACC: Post Transfer Phase
ACC: Trans 1
ACC: Trans 2
ACC: Trans 3
ACC:     Simple transfer of 'var%comp_b' (72 bytes)
ACC:         host ptr 4c0f00
ACC:         acc  ptr 0
ACC:         flags: REG_PRESENT OMP_PTR_ATTACH
ACC:         host region 4c0fc0 to 4c0fc1 found in
present table index 0 (ref count 1)
ACC:             attach pointer host 0x4c0f00 (pointee
0x4c0fc0) to device 154e54427000 (pointee 154e54409000)
for 'var%comp_b' from origMain.F90:28
ACC:             internal copy host to acc (host d38b70
to acc 154e54427000) size = 72
ACC:
ACC: End transfer (to acc 120000 bytes, to host 0 bytes)
ACC:
```

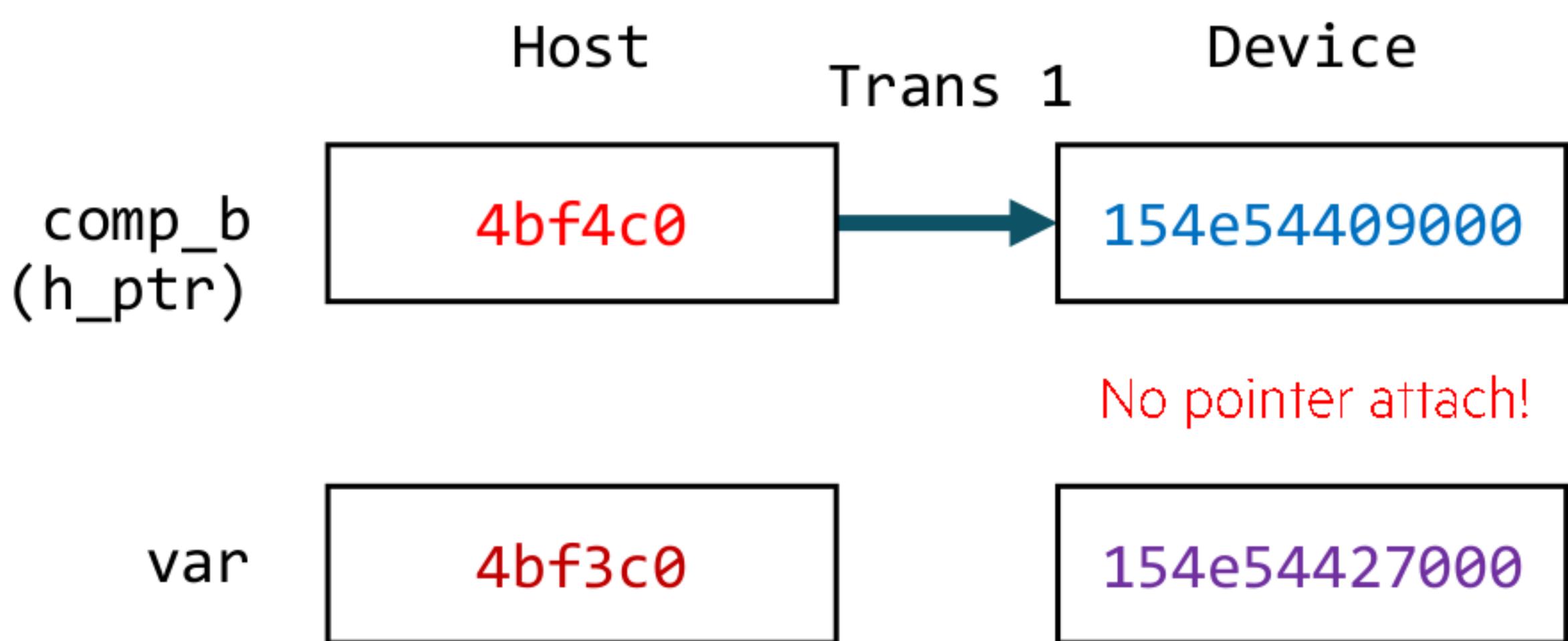


APPLICATION USE CASE

Mapping Derived Types with Pointer components, **not inlined**

```
ACC: Start transfer 1 items from origMain.F90:13  
ACC:   flags:  
ACC:  
ACC:   Trans 1  
ACC:     Simple transfer of 'h_ptr(:)' (72 bytes)  
ACC:       host ptr 4bf3c0  
ACC:       acc  ptr 0  
ACC:       flags: DOPE_VECTOR DV_ONLY_DATA ALLOCATE  
COPY_HOST_TO_ACC ACQ_PRESENT REG_PRESENT  
ACC:       Transferring dope vector  
ACC:         DV size=120000 (dim:1 extent:30000  
stride_mult:1 scale:4 elem_size:4)  
ACC:           total mem size=120000 (dv:0 obj:120000)  
ACC:           memory not found in present table  
ACC:           allocate (120000 bytes)  
ACC:             get new reusable memory, added entry  
ACC:             new allocated ptr (154e56a09000)  
ACC:             add to present table index 0: host 4bf4c0 to 4dc980, acc 154e56a09000  
ACC:             copy host to acc (4bf4c0 to 154e56a09000)  
ACC:               internal copy host to acc (host 4bf4c0 to acc 154e56a09000) size = 120000  
ACC:             new acc ptr 154e56a09000  
ACC: End transfer (to acc 120000 bytes, to host 0 bytes)  
ACC: Start transfer 1 items from origMain.F90:30
```

```
8 subroutine map_array(h_ptr)  
9   implicit none  
10  integer, pointer :: h_ptr(:)  
11  
12  !$omp target enter data map(to:h_ptr)  
13  
14  end subroutine map_array  
15 end module test_map
```



(This transfer happens later)

APPLICATION USE CASE

Mapping Derived Types with Pointer, continued.

- A workaround was found
- Additional OpenMP map was required to perform the pointer attachment to the derived type on the device
- The data transfer is still happening at the map clause on line 13 (non-inlined case)
 - The pointer attachment is happening on line 30
- Unfortunately, this scheme does not “hide” all of the OpenMP data directives within the subroutine call.
 - The ideal case being that we could omit the additional map directive on line 30

```
1 module test_map
2   type type_a
3     integer, pointer, contiguous :: comp_b(:)
4   end type type_a
5
6 contains
7
8   subroutine map_array(h_ptr)
9     implicit none
10
11   integer, pointer :: h_ptr(:)
12
13   !$omp target enter data map(to:h_ptr)
14 end subroutine map_array
15 end module test_map
16
17 program test_mapper
18 !DIR$ NOINLINE
19 use test_map
20 implicit none
21 integer, parameter :: n=30000
22 integer :: i
23
24 type(type_a), allocatable:: var
25 allocate(var)
26 allocate(var%comp_b(n))
27
28 call map_array(var%comp_b)
29
30 !$omp target teams distribute simd map(tofrom:var%comp_b)
31 do i=1,n
32   var%comp_b(i) = i
33 end do
34
35 end program test_mapper
```

PUTTING IT ALL TOGETHER

“Forcing” data movements

- The rules for data movements can get rather complicated and transfers may not occur when they were expected
 - The reference counts for the variables can get unexpectedly higher than expected
- We often find that we conceptually expect every stand-alone map clause to be impactful and perform data movements
 - No-ops are typically not expected
- We will examine an example from the FGPU repo that has more advanced data structures and tries to map things over
 - I've condensed most of the code in the subsequent slides
 - The exact example is from FGPU/openmp/target_associate_ptr_under_an_api/nested_data



NESTED DATA EXAMPLE

Data structures used

```
type, public :: typeS
    real(C_DOUBLE)                      :: double
    real(C_DOUBLE), pointer, dimension(:) :: double_array
    real(C_DOUBLE), pointer, dimension(:, :) :: double_array_2d
    real(C_DOUBLE), pointer, dimension(:, :, :) :: double_array_3d
end type typeS

type, public :: typeG
    real(C_DOUBLE)                      :: double
    real(C_DOUBLE), pointer, dimension(:) :: double_array
end type typeG

type, public :: typeQ
    real(C_DOUBLE)                      :: double
    real(C_DOUBLE), pointer, dimension(:) :: double_array
    type(typeS),   pointer, dimension(:) :: s_array
    type(typeG),   pointer, dimension(:) :: g_array
end type typeQ

type(typeQ), pointer, public :: typeQ_ptr
```

NESTED DATA EXAMPLE

Initializing and mapping to the device

```
call initialize() ! All arrays are allocated and set to the value "1". The s_array has two elements
! Mapping to the device
 !$omp target enter data map(to:typeQ_ptr)
 !$omp target enter data map(to:typeQ_ptr%s_array(1)%double_array)
 !$omp target enter data map(to:typeQ_ptr%s_array(1)%double_array_2d)
 !$omp target enter data map(to:typeQ_ptr%s_array(1)%double_array_3d)
 !$omp target enter data map(to:typeQ_ptr%s_array(2)%double_array)
 !$omp target enter data map(to:typeQ_ptr%s_array(2)%double_array_2d)
 !$omp target enter data map(to:typeQ_ptr%s_array(2)%double_array_3d)

! Doing work on the device
 !$omp target
 typeQ_ptr%double = 0
 typeQ_ptr%double_array = 0
 do n=1,2
   typeQ_ptr%s_array(n)%double = 0
   typeQ_ptr%s_array(n)%double_array = 0
   typeQ_ptr%s_array(n)%double_array_2d = 0
   typeQ_ptr%s_array(n)%double_array_3d = 0
 enddo
 !$omp end target

! Getting data back from the device
 !$omp target exit data map(from:typeQ_ptr%s_array(2)%double_array)
 !$omp target exit data map(from:typeQ_ptr%s_array(2)%double_array_2d)
 !$omp target exit data map(from:typeQ_ptr%s_array(2)%double_array_3d)
 !$omp target exit data map(from:typeQ_ptr%s_array(1)%double_array)
 !$omp target exit data map(from:typeQ_ptr%s_array(1)%double_array_2d)
 !$omp target exit data map(from:typeQ_ptr%s_array(1)%double_array_3d)
 !$omp target exit data map(from:typeQ_ptr)

call print_values(typeQ_ptr)
```

NESTED DATA EXAMPLE

Adding the always map-identifier to the map clause

```
call initialize() ! All arrays are allocated and set to the value "1". The s_array has two elements
! Mapping to the device
 !$omp target enter data map(always,to:typeQ_ptr)
 !$omp target enter data map(always,to:typeQ_ptr%s_array(1)%double_array)
 !$omp target enter data map(always,to:typeQ_ptr%s_array(1)%double_array_2d)
 !$omp target enter data map(always,to:typeQ_ptr%s_array(1)%double_array_3d)
 !$omp target enter data map(always,to:typeQ_ptr%s_array(2)%double_array)
 !$omp target enter data map(always,to:typeQ_ptr%s_array(2)%double_array_2d)
 !$omp target enter data map(always,to:typeQ_ptr%s_array(2)%double_array_3d)

! Doing work on the device
 !$omp target
 typeQ_ptr%double = 0
 typeQ_ptr%double_array = 0
 do n=1,2
   typeQ_ptr%s_array(n)%double = 0
   typeQ_ptr%s_array(n)%double_array = 0
   typeQ_ptr%s_array(n)%double_array_2d = 0
   typeQ_ptr%s_array(n)%double_array_3d = 0
 enddo
 !$omp end target

! Getting data back from the device
 !$omp target exit data map(always,from:typeQ_ptr%s_array(2)%double_array)
 !$omp target exit data map(always,from:typeQ_ptr%s_array(2)%double_array_2d)
 !$omp target exit data map(always,from:typeQ_ptr%s_array(2)%double_array_3d)
 !$omp target exit data map(always,from:typeQ_ptr%s_array(1)%double_array)
 !$omp target exit data map(always,from:typeQ_ptr%s_array(1)%double_array_2d)
 !$omp target exit data map(always,from:typeQ_ptr%s_array(1)%double_array_3d)
 !$omp target exit data map(always,from:typeQ_ptr)

call print_values(typeQ_ptr)
```

The transfer to the host is explicitly requested here regardless of the ref. count

MAIN TAKEAWAYS

- Derived types on the GPU need to have their pointer components properly “attached” on the GPU
 - If you plan on referencing pointer components through their derived types in target regions, they must appear on a map clause as a list item fully qualified from the derived type base
- Be explicit with your data structure movement
 - Don’t rely on implicit behaviors if you can avoid it
 - **default(none)** is your friend especially when debugging
- Use the tools available in the standard!
 - For example: If you absolutely need something moved from the GPU use the **always** modifier in the **map** clause
 - The **always** modifier for the **map** clause tends to be used intended in the standalone directives
- The OpenMP API has the tools to integrate external allocators to your application
- Be ready to change things as new features are added to the OpenMP standard
 - Work arounds that you had to do for previous specifications may not be needed anymore
 - The 5.0 standard has important support that facilitates the implementation and portability of OpenMP offload
 - Vendors are always interested in the use cases of real applications



THANK YOU

QUESTIONS?

