OpenMP Offloading Sessions

- *HPC GPU Computing*
  - GPU Architectures
- Offload basics

`git clone https://github.com/csc-training/advancedOpenMP.git`

  - Fundamentals
  - OpenMP Target Offload Directive
- Data Management (movement)
  - Mapping variables
  - Mapping pointers and pointee data, alloc map-type
  - Functions and Static Variable in external Compile Units
  - Persistence/Updating
  - Direct Allocation on Devices
- Advanced Features and more on teams distribute parallel
  - Mapper
  - Async Offloading
  - function variants
  - target teams distribute parallel for|do

```
git clone https://github.com/
          csc-training/advancedOpenMP.git
```

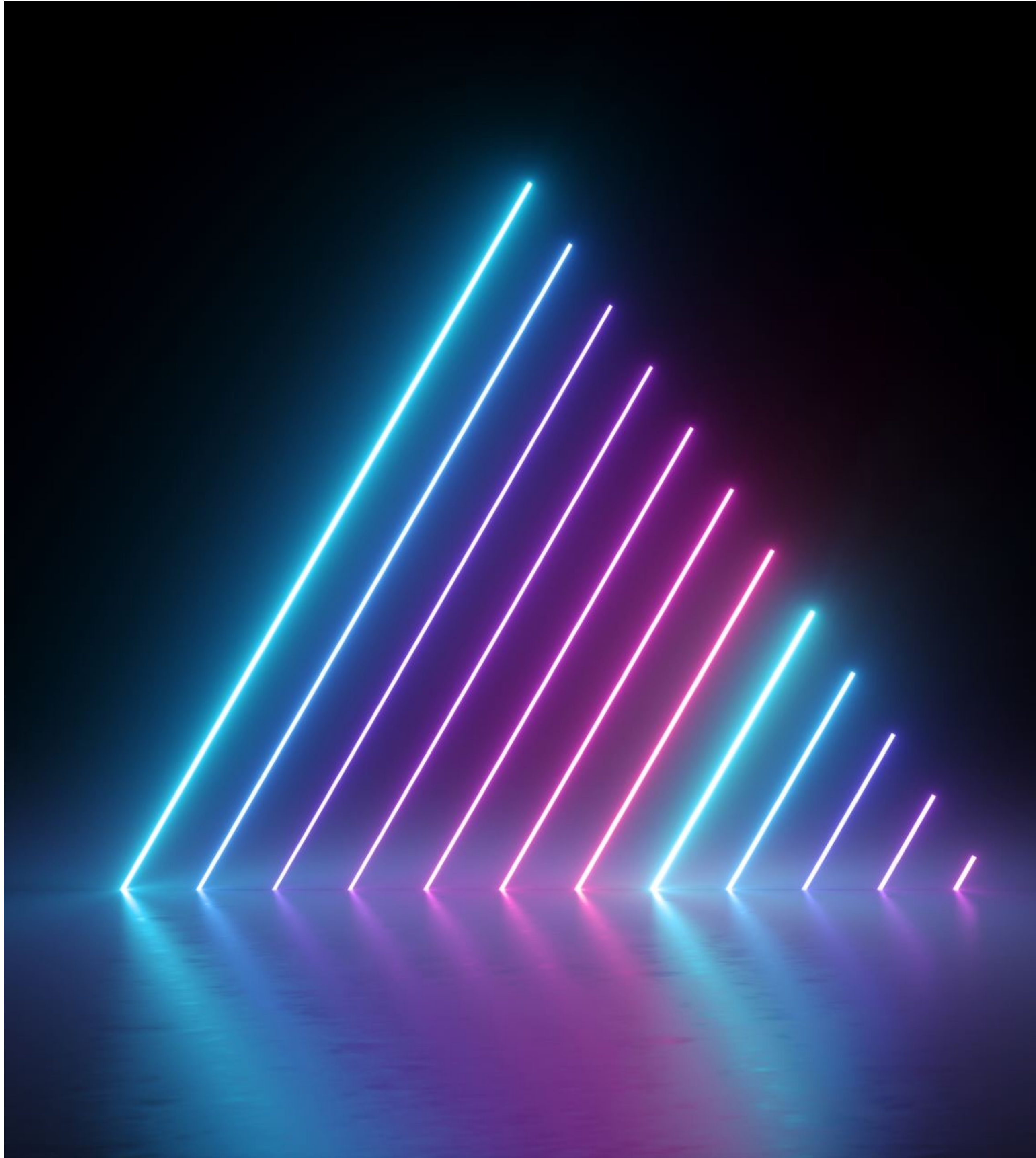
OpenMP Offloading Data Management

by

Kent Milfeld (TACC)

Emanuele Vitali (CSC)

slides: tinyurl.com/tacc-csc-2023-offload



Section Objective

- OpenMP Offloading
  - Learn what mapping is
  - How and when implicit (automatic) transfer/storage management occurs.
  - How to explicitly transfer and manage storage through OpenMP map clause.
  - How to declare device function
  - Structured/unstructured device storage/data persistence.

Summary: Target execution on GPU

- **target**: execute next function, statement or code block on GPU
- **teams**: use the SMs
- **distribute parallel for**: distribute *for* iterations across SMs & CUs and workshare with parallel for

```
int main(){
   int n = 1<<28;
   float x[n], y[n], a=2.0f;
   init(n,x,y);

#pragma omp target teams distribute parallel for
   for(int i=0; i<n; i++) y[i] = a*x[i] + y[i];

}
```

Will often not include `teams distribute parallel for` in following slides, or use "...".

OpenMP Data Motion in Offloading

- Device Directives -- since OpenMP 4.0 (July 2013)
- Things have evolved significantly since then in:
  data motion, mem alloc, async behavior, parallel execution,...

- Data section will be prefaced with:
  Execution Model
  Host/Device Data terminology
  Scoping Info
  Memory Management -- mapping

OpenMP Execution Model for Offloading

- When a **target** construct is encountered, a new *target task* is generated.
- The target task region encloses the target region.
- The target task is complete after the execution of the target region is complete.

relevant to async behavior in later slides

OpenMP Memory and Memory Management

- A host CPU may use a separate memory space from a GPU device
- Unified Shared Memory (USM) is memory which shares access between different devices (CPU and GPU). USM is not discussed here.
- OpenMP supports both USM and separate memory spaces

- OpenMP will automatically (implicitly) allocate space and transfer host data to/from the device (when variables are within "scope").

- Explicit methods (constructs, clauses and API routines) are used to control device storage and data transfer to/from.

OpenMP Terminology: data environment

- A data environment exists for the task encountering a target region.
- A data environment for the target device is created.

```
int main(){
  int a[2]={1,2};

  #pragma omp target
  { a[0]=1; a[1]=2;}
}
```

Original Variable

Data Environment
of task executing main
(& encounters target)

Data Environment
of target device

Corresponding (device)
Variable

```
program main
  integer :: a(2)=[1,2]

  !$omp target
   a(1)=1; a(2)=2;
  !$omp end target
end program
```

OpenMP Scoping Terminology

- **target lexical extent** of directive: what compiler sees in statement/code block (but not inside functions called).
- **target region** of directive: lexical extent + inside functions.

```
fun(int *a, int *b){*a+=1; *b+=2;}              target
                                                region
int main(){
   int a,b;

   #pragma omp target  map(a,b)
   {
     a=1;                           target
     b=2;                           lexical     target
     fun(&a,&b);                    extent      region
   }
}
```

Outline

- Data Motion
    - Mapping variables
    - Mapping pointers and pointee data, array sections, alloc map-type
    - Functions and Static Variables in External Compile Units
    - Structured/Unstructured Mapping and Update Directives
    - Direct Allocations on Devices

OpenMP Terminology: map clause

- a **map clause** is used to specify how original variables are "mapped" to corresponding variables in the device data environment of a target region.

- Creating data that **persist across multiple targets** (within a structured block) is handled by "structured" data mapping constructs, and creation/updates at **any location** within the program by "unstructured" data mapping **constructs**.
- **API routines can allocate device storage** (like cudamalloc).

More on these later.

Mapping – basics: allocation and copy

- Mapping Operations
  - allocating device storage for corresponding variables on device
  - transferring original variable data to & from corresponding variable of device
  - deleting device storage

- implicit mapping:
  - automatic variable mapping

- explicit mapping: with map clause:
  - minimizes unnecessary data transfers of implicit mapping
  - necessary to specify the extent (how much) of dynamic host memory to map
  - apply other controls on mapping (mappers, etc....)

Implicit Mapping -- automatic allocation and copy

data variables within the **lexical extent** of a target region:

At the **beginning** of the target region, storage for the variables is allocated and the original variable is copied to the corresponding item on the device.*

At the **end**, corresponding variables are copied from the device and deallocated.*

implicit mapping

* Exception, scalars are firstprivate.

```
                                alloc & copy x,y,n,a to dev; n,a are firstprivate
#pragma omp target
  for(int i=0;i<n;i++) y[i]=a*x[i]+y[i];  } target lexical extent
                                dealloc & copy x,y from device to host
```

Spec 5.2 pg 148 Implicit Data-Mapping Attribute Rules

Explicit Mapping map Clause – simple usage

Syntax*:

```
map( [map-type :] variable list )
```

*Simple, incomplete syntax.

variable list

`variables` comma separated list of variables – “list items”

map-type

`alloc`	allocate storage for corresponding variable
`to`	`alloc` and assign value of original variable to corresponding variable on target entry
`from`	`alloc` and assign value of corresponding variable to original variable on exit
`tofrom`	default, both `to` and `from`

Explicit Mapping map Clause – simple usage

```
#pragma omp target ... map(tofrom: y,x) map(to: n,a)
  for(int i=0;i<n;i++) y[i]=a*x[i]+y[i];
```

Effectively same behavior as previous implicit map and firstprivate data attribute.

Variable **list items**:

	Base Language Examples C/C++	F90
scalars	**int a;**	**integer :: a**
arrays	**int a[9];**	**integer :: a[9]**
pointers	**int*ptr;**	**integer, pointer :: a**
objects (structures)	**struct ...**	**data type ...**
	OpenMP-defined for map	
array sections	**a[***start:extent,stride***]**	**a(:::)**

- Important quark: **scalars are not implicitly mapped but have firstprivate data attribute**. Use map clause to avoid firstprivate on specific variables, or defaultmap(tofrom: scalar) for all scalars on target construct.

Complete **map** clause syntax/description

`map([[map-modifier,...] map-type:] locator-list)`

"locator*" items are:
lvalue* expressions including variables, array sections, omp reserved locators

map-modifier

`always`	map always occurs	**`mapper`** (*mapper-id*)	use a defined map
`close`	alloc close to device	`iterator` (*iters-def*=range)	
`present`	on entry, terminate if corresponding variable not present **		

map-type

`alloc, to, from, tofrom`	
`delete`	corresponding variable is removed
`release`	reference count is decremented

*object with an identifiable location in memory ** **if** clause may preclude entry

mapping directives

- The map clause is also used for (explained later):
  - persistent data directives
    target data
    target enter data
    target exit data
  - user-defined (shorthand) of map clause(s)
    declare mapper

setting default mapping on target construct

`defaultmap(`*`implicit-behavior`*` [`***`:variable-category`***`])`

a target clause

variable-category

scalar	(C/C++, Fortran)
aggregate	(structures, classes, derived types)
pointer	(C/C++,Fortran)
allocatable	(Fortran)

implicit-behavior

to, from, tofrom, alloc, default, **none**, **firstprivate**

Outline

- Data Motion
  - Mapping variables
  - Mapping pointers and pointee data, array sections, alloc map-type
  - Functions and Static Variables in External Compile Units
  - Structured/Unstructured Mapping and Update Directives
  - Direct Allocations on Devices

Explicit Mapping -- Array Sections

- C/C++ Pointers are extremely versatile, but the base language has no way to express an "extent" of data pointed to
- Mapping needs an extent for storage creation
- OMP defines an **array section** with a variable name and a range:

Syntax: `var[ lower-bound : length : stride ]`

```
int * x=(int*)malloc(n*sizeof(n));
int * y=(int*)malloc(n*sizeof(n));

#pragma omp target ... map(tofrom: y[0:n]) map(to: x[0:n])
   for(int i=0;i<n;i++) y[i]=a*x[i]+y[i];
```

Explicit Mapping -- Array Sections

- Array Section:
  - All or subsection: **[lower-bound : length : stride]**
  - Works with multidimensional arrays
  - Multiple non-overlapping maps are OK.
- integer expressions are allowed for arguments

– **lower-bound** defaults to 0

– **length** must be explicitly specified when array dimension not known

– **stride** defaults to 1 (must positive)

```
map(tofrom: a[0:n/4) a[n/2:n/4])
```

```
map(tofrom: a[0:n/2) map(tofrom: a[n/2:n/4])
```

Explicit Mapping – **alloc** for temp space (for host & dev)

- Map-type ***alloc*** can be used just to allocate space (no copy)

More later.

C/C++

```
#pragma omp target ... map(alloc: x) map(tofrom: y)
{
   for(int i=0;i<n;i++) x[i]=sin(i%4)*cos(i%6);

   for(int i=0;i<n;i++) y[i]=a*x[i]+y[i];
}
```

Allocated only for this target region.

Outline

- Data Motion
  - Mapping variables
  - Mapping pointers and pointee data, array sections, alloc map-type
  - Functions and Static Variables in External Compile Units
  - Structured/Unstructured Mapping and Update Directives
  - Direct Allocations on Devices

Creating Device Functions

functions defined and used within the same scope are automatically compiled for the device (GPU).

```
  void fun( int i, float a, float *x, float *y )
                              { y[i] = a*x[i] + y[i]; }
int main(){
...
#pragma omp target teams distribute parallel for
  for(int i=0; i<n; i++) fun(n,a,x,y);
```

Functions defined **outside of file scope require declaration:**

```
void fun( int i, float a, float *x, float *y )
                              { y[i] = a*x[i] + y[i]; }
#pragma omp declare target (fun)
```

declare target syntax: (basic)

```
#pragma omp begin omp declare target
   //procedures & static variables here
#pragma omp end omp declare target
```

C/C++ only

```
#pragma omp declare target (ext_list)
```

Use !$omp for F90

```
    ext_list: list of procedures(pcr) and/or static variables(var)
```

```
#pragma omp declare target clauses
```

Use !$omp for F90

```
   clauses:  to/enter(ext_list)     avail. entire prog
             link(     var_list)    avail. when mapped
             device(expr)           device specific (ids)
             indirect(pcr_list)     4 func. pointers
```

to(<v5.2) enter(>=v5.2)

declare target -- C/C++ examples

```
#pragma omp begin declare target                          C/C++

   static   float a_global=2.0f;
   function foo(){printf("external compiled unit\n");}

#pragma omp   end declare target
```

This is for declaring everything within a region

```
static   float a_global=2.0f;                             C/C++
function foo(){printf("external compiled unit\n");}

#pragma omp declare target (a_global, foo)
```

This is for declaring specific vars and funcs.

```
static   float a_global=2.0f;                             C/C++
function foo(){printf("external compiled unit\n");}

#pragma omp declare target to (a_global, foo)
```

Same as above, using clause (***to*** is default).

declare target – Fortran examples

```
subroutine soo()                                                F90
    print*, "externally compiled"
    !$omp declare target                 ! implied enter(soo) – no clauses
!or !$omp declare target enter(soo)      ! use "to" (<v5.2), use other clauses
end subroutine
```

```
module global_v                                         F90
    integer,parameter    :: N=100
    real                 :: v(N)
    !$omp declare target   (v)      !implied enter
!or !$omp declare target to(v) !can use other clauses here
   contains
    subroutine init()
    !$omp declare target                   !implied enter(init)
!or !$omp declare target to(init)  !can use enter() for >=v.52
      v=1
    end subroutine
end module
```

Outline

- Data Motion
  - Mapping variables
  - Mapping pointers and pointee data, array sections, alloc map-type
  - Functions and Static Variable in External Compile Units
  - Structured/Unstructured Mapping and Update Directives
  - Direct Allocations on Devices

Structured/Unstructured Mapping & Update Definitions

- Implicit/Explicit mapping on a single **target** construct is convenient
- However, other mapping-only directives are available for:

- persistence over a region	structured	**target data**
- not region specific	unstructured	**target enter data**
		target exit data

- Update directive (non-mapping)

- makes corresponding list item (var) consistent with original item (var)	**target update**

target data, enter data, and exit data directives

- A structured mapping encloses one or more target regions.
- Unstructured mapping can occur anywhere.

Syntax:

```
target       data map(...) [clauses]
target enter data map(...) [clauses]
target exit  data map(...) [clauses]
```

clauses

map in background → **nowait**
order mapping(s)* → **depend()**
map for specific device(s) → **device()**
conditionally execute → **if()**

*data motion constructs are executed by ***target tasks***, allowing task async behavior.

target update directive

- Makes corresponding & original variables consistent via motion-clause

Syntax: `target update data-motion_clause(s) [clauses]`

data-motion-clause

update device data → **to**
update host data → **from**

clauses

map in background → **nowait**
order mapping(s)* → **depend()**
map for specific device(s) → **device()**
conditionally execute → **if()**

*data motion constructs are executed by ***target tasks***, allowing task async behavior.

target data – target data regions

C/C++

```
int main () {
   int A[2]={10,11};


   #pragma omp target data map(tofrom: A)  // map applies to the region {}
   {
         #pragma omp target    // Data is already on device (ref cnter keeps track)
         {                     // No implicit map, reference counters keeps track
             A[0]++; A[1]++;
         }

         #pragma omp target    // Data is already on device
         {
             A[0]++; A[1]++;
         }

   }                            // A copied back to host and freed from device
}
```

target data – target data regions with update

C/C++

```
int main () {
   int A[2]={10,11};


   #pragma omp target data map(tofrom: A[:2])
   {

         A[0]=0; A[1]=1;
         #pragma omp target update to(A[:2])


         #pragma omp target     // implicit map does nothing, bc A is already present
         {                      //                (A[0]:A[1])
             A[0]++; A[1]++;    //   1   :  2     on device
         }                      //   0   :  1     inside data region

   }
   //                           //   1   :  2     outside data region
}
```

target enter data, and exit data directives

- Unstructured mapping can occur anywhere.

Syntax:

```
target enter data map(map-type: ...) [clauses]
target exit  data map(map-type: ...) [clauses]
```

	map type (required)	*clauses*
target enter data →	**to, alloc**	**nowait**
target exit data →	**from, release, delete**	**depend()**
		device()
		if()

target enter/exit data -- simple example

C/C++

```
#include <iostream>
#include <omp.h>
using namespace std;


int main(){
 int A[2]={10,11};

  #pragma omp target enter data map( to: A)


     #pragma omp target
     {A[0]++; A[1]++;}


  #pragma omp target exit data map(from: A)
  cout << A[0]<< "  " << A[1] <<endl;
}
```

target enter/exit data -- example

C/C++

```
class Vec{
public:
   Vec(int n) : len(n){
                        v = new double[len];
                        #pragma omp target enter data map( alloc:  v[0:len] )}

  ~Vec()              {
                        #pragma omp target  exit data map( delete: v[0:len] )
                        delete[] v;   }

   double *v; int len;
};
int main(){
  int n=16; Vec cl_v(n);

  #pragma omp target map(tofrom:cl_v.v[0:n])
  for(int i=0; i<n; i++) cl_v.v[i]=11.0f;

  #pragma omp target update from(cl_v.v[0:n])
  for(int i=0; i<n; i++) cout << i << " " << cl_v.v[i] << endl;
}
```

target data — scalar firstprivate

C/C++

```
int Fact = 1;
#pragma omp target data map(tofrom: Fact)   //  Fact is mapped for region
{
    Fact=10;
    #pragma omp target map(always,tofrom: Fact) // make device/host consistent (10)
    { Fact++;                                   // mapped Fact =  11
    }                                           // cp back Fact = 11

    Fact++;
    #pragma omp target // default Fact is FIRSTPRIVATE use Host Fact (12)
    { Fact++;          // 13
    }                  // 12 Fact Not Copied back

    Fact=100;
    #pragma omp target update to(Fact)   // update mapped Fact (100 on host/dev)
    Fact=500;
    #pragma omp target map(Fact)         // use mapped Fact (no consistency updating)
    {   Fact++;                          // 101
    }                                    // 100 No cp back (only "from" modifier)
}
```

Outline

- Data Motion
  - Mapping variables
  - Mapping pointers and pointee data, array sections, alloc map-type
  - Functions and Static Variables in External Compile Units
  - Structured/Unstructured Mapping and Update Directives
  - Direct Allocations on Devices

omp_target_alloc syntax

Syntax: void* omp_target_alloc(size_t size, int device_num);

returns the device address of
a storage location of size bytes

device_num : less than the result of
omp_get_num_devices() or the result of a call to
omp_get_initial_device().

```
   int init_dev =omp_get_initial_device();
   int     ndevs=omp_get_num_devices();
   cout<<"init_dev#= "<<init_dev <<" # non-host_devs"<< ndevs<<endl;

OUTPUT: init_dev#= 3 # non-host_devs3
```

omp_target_alloc() + is_device_ptr

C/C++

```
int main(){
int a=2, N=1<<4; // 16

  int *y   =(int*)           malloc(N*sizeof(N) );
  int *x_d=(int*)omp_target_alloc(N*sizeof(N),omp_get_default_device());

  for(int i=0;i<N;i++){ y[i]=1; }

  #pragma omp target is_device_ptr(x_d)
  for(int i=0;i<N;i++) x_d[i]=1;           //INIT

  #pragma omp target is_device_ptr(x_d) \
                     map(tofrom: y[0:N])
  for(int i=0;i<N;i++) y[i]=a*x_d[i]+y[i]; //AXPY

}
```

Other Device Memory Routines

C/C++	Fortran	Description
int omp_target_is_present(const void *ptr, int device_num);	integer(c_int) function omp_target_is_present(ptr, device_num) & bind(c) use, intrinsic :: iso_c_binding, only : c_ptr, c_int ..	routine tests whether a host pointer refers to storage that is mapped to a given device.
int omp_target_is_accessible(const void *ptr, size_t size, int device_num);	integer(c_int) function omp_target_is_accessible(& ptr, size, device_num) bind(c) use, intrinsic :: iso_c_binding, only : c_ptr, c_int ..	routine tests whether host memory is accessible from a given device.
int omp_target_memcpy(void *dst,..);	**integer(c_int) function omp_target_memcpy(dst, src, length, & dst_offset, src_offset, dst_device_num, src_device_num) bind(c) use, intrinsic :: iso_c_binding, only : c_ptr, c_int, c_size_t ..**	**routine copies memory between host and device pointers.**

Other Device Memory Routines (cont.)

C/C++	Fortran	Description
int omp_target_memcpy_rect(..);	integer(c_int) function omp_target_memcpy_rect(dst,src,element_size, & .	copies a rectangular sub-volume from a multi-dimensional array to another multi-dimensional array.
int omp_target_memcpy_async(...);	integer(c_int) function omp_target_memcpy_async(..	performs asynchronous copy between host and device pointers.
int omp_target_memcpy_rect_async(...);	integer(c_int) function omp_target_memcpy_rect_async(...	asynchronously performs a copy between host and device pointers.
int omp_target_associate_ptr(...);	integer(c_int) function omp_target_associate_ptr(...	routine maps a device pointer to a host pointer
int omp_target_disassociate_ptr(...);	integer(c_int) function omp_target_disassociate_ptr(..	routine removes the associated pointer for a given device from a host pointer.
void * omp_get_mapped_ptr (...);	type(c_ptr) function omp_get_mapped_ptr(...	routine returns the device pointer that is associated with a host pointer for a given device.

omp_target_memcpy

C/C++

```
void get_dev_cos(double *mem, int s){
   int h, t, i;
   double * mem_dev_cpy;
   h = omp_get_initial_device();
   t = omp_get_default_device();

   mem_dev_cpy = (double *)omp_target_alloc( sizeof(double) * s, t);

                         /* dst   src */
   omp_target_memcpy(mem_dev_cpy, mem, sizeof(double)*s,
                             0,     0,
                             t,    h);

   #pragma omp target is_device_ptr(mem_dev_cpy) device(t)
   #pragma omp teams distribute parallel for
     for(i=0;i<s;i++){ mem_dev_cpy[i] = cos((double)i); } /* init data */

                  /* dst   src */
    omp_target_memcpy(mem, mem_dev_cpy, sizeof(double)*s,
                      0,              0,
                      h,              t);
    omp_target_free(mem_dev_cpy, t);
}
```

Map: Reference count

- On entry to device environment:
  - If a corresponding list item is not present in the device data environment, then:
    - A new list item corresponding to original list item (on host) is created in the device data environment;
    - The corresponding list item has a reference count that is initialized to zero; and
    - The value of the corresponding list item is undefined;
  - If ref count is not incremented due to map clause, it is incremented by 1

Atomic Operation

- On exit from device environment:
  - if map-type is **delete** ref count is set to 0
  - if map-type is not **delete** the ref count is decremented by 1 (min 0)
  - If the reference count is zero then the corresponding list item is removed from the device data environment.

Atomic Operation