Recap: OpenMP Offload

• OpenMP offload constructs are a set of directives for C, C++, and Fortran that were introduced in OpenMP 4.0 and further enhanced in later versions. Accelerators
Recap: target directive

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp target [clause[ [,] clause] ... ] new-line structured-block</td>
<td>!$omp target [clause [ [,] clause] ... ] loosely/tightly-structured-block !$omp end target</td>
<td>The target construct offloads the enclosed code to the accelerator.</td>
</tr>
</tbody>
</table>

- A device data environment is created for the structured block
- The code region is mapped to the device and executed.
Recap: Clauses on target directive

- Clauses allowed on the target directive:
  - `device([ device-modifier :] integer-expression)`
  - `if([ target :] scalar-expression)`
  - `thread_limit(integer-expression)`
  - `private(list)`
  - `firstprivate(list)`
  - `in_reduction(reduction-identifier : list)`
  - `map([[map-type-modifier[,] [map-type-modifier[,] ...]] map-type: ] locator-list)`
  - `is_device_ptr(list)`
  - `has_device_addr(list)`
  - `defaultmap(implicit-behavior[:variable-category])`
  - `nowait`
  - `depend([depend-modifier[,] dependence-type : locator-list])`
  - `allocate([allocator :] list)`
  - `uses_allocators(allocator[[allocator-traits-array]] [,allocator[[allocator-traits-array]] ...])`
map clause on target directive

Syntax: \texttt{map([map-type-modifier[,] [map-type-modifier[,] ...] map-type : ] locator-list)}

“The map clause specifies how an original list item is mapped from the current task’s data environment to a corresponding list item in the device data environment of the device identified by the construct.”
map clause: map-type-modifiers

- **always**
  - value of list item is always copied to (for **to** and **tofrom**) and from device (for **from** and **tofrom**)

- **close**
  - **hint** to the runtime to allocate memory close to the target device

- **present**
  - sets a requirement that the corresponding list item already exists in the device data environment
  - If the list item is not present it causes a **runtime error**
map clause: other modifiers

- **mapper**
  - Provides a mechanism to override implicit mapping and provide custom mapping
  - A `declare mapper` directive must be used to create a unique mapper
  - Use case: Nested structure elements or nested structures (deep copy)

```c
/*C code to show use of mappers*/
...
typedef struct S{
    size_t len;
    double *data;
} S_t;

#pragma omp declare mapper(X: S_t s) map(s, s.data[0:s.len])
...
int main(){
    S_t A;
    //allocate and initialize elements of A
    #pragma omp target map(mapper(X))
    {
        //work using array elements of A
    } // end target
    ...
```
map clause: other modifiers (cont.)

• iterator
  – Defines a set of iterators, each of which is an iterator-identifier and an associated set of values
  – Expands based on the values assigned
  – Example:
    • map(int iterator(i=0:n), tofrom: p[i])
map clause: map-types

- to
  - allocates data and moves data to the device

- from
  - allocates data and moves data from the device

- tofrom
  - allocates data and moves data to and from the device

- alloc
  - allocates data on the device

- release
  - decrements the reference count of a variable by 1

- delete
  - reference count of a variable is set to 0
  - deletes the data from the device

If a map-type is not specified, the map-type defaults to tofrom
Mapping Rules: Reference Count

• On entry to device environment:
  • If a corresponding storage block is not present in the device data environment, then:
    • A new storage block corresponding to original list item (on host) is created in the device data environment;
    • Reference count of this storage block is initialized to zero; and
  • The ref count is then incremented by 1
  • For every list item in the storage block
    • If ref count of the storage block is 1 - new list item is created in the storage block
    • If ref count of the list item is 1 or always map-type-modifier is present, and map-type is to or tofrom the list item value on the target device is updated to the value on the host device
Mapping Rules: Reference Count (cont.)

• On exit from device environment:
  – If ref count is 1 or always map-type-modifier is specified, and map-type is from or to from original list item (on the host device) is updated
  • if ref count is finite and:
    • map-type is delete ⇒ ref count is set to 0
    • if map-type is not delete ⇒ 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.
Implicit Mapping Rules on target directive

- C/C++: Pointer is treated as if it is the base pointer of a zero-length array section mapped using the map clause.

- Fortran: If a scalar variable has the TARGET, ALLOCATABLE or POINTER attribute treated \( \rightarrow \) map( tofrom: )

- All:
  - Scalars variables are implicitly firstprivate (not mapped)
  - If a variable is not a scalar then it is treated as if it was mapped with a map-type of tofrom.
Implicit Mapping Rules on target directive (cont.)

- A base variable ‘X’ of a list item ‘a’ is in a reduction, lastprivate or linear clause on a combined target construct → treated as if \text{map(tofrom: X)}

- A base variable ‘X’ of a list item ‘a’ is in an in_reduction clause on a target construct → treated as if \text{map(always, tofrom:a)}
Allocating Memory on the Target Device

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void* omp_target_alloc(size_t size, int device_num);</td>
<td>type(c_ptr) function omp_target_alloc(size, device_num) bind(c) use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t, c_int integer(c_size_t), value :: size integer(c_int), value :: device_num</td>
<td>routine allocates memory in a device data environment and returns a device pointer to that memory.</td>
</tr>
<tr>
<td>void omp_target_free(void *device_ptr, int device_num);</td>
<td>subroutine omp_target_free(device_ptr, device_num) bind(c) use, intrinsic :: iso_c_binding, only : c_ptr, c_int ..</td>
<td>routine frees the device memory allocated by the omp_target_alloc routine.</td>
</tr>
</tbody>
</table>

- The `omp_target_alloc` routine returns a device pointer that references the device address of a storage location of size bytes.

- The storage location is dynamically allocated in the device data environment of the device specified by `device_num`. 
is_device_ptr clause on target directive

• The is_device_ptr clause indicates that its list items are device pointers
• For C++ the list item must be:
  • type of pointer or array,
  • reference to pointer or reference to array
• For C it must have a type of pointer or array.
• For Fortran the list item must be of type C_PTR
• Support for device pointers created outside of OpenMP is implementation defined
• is_device_ptr clause is not necessary when using requires unified_address
Using is_device_ptr

/*C code for example of is_device_ptr*/

int *array_device = NULL;
int *array_host = NULL;

array_device = (int *) omp_target_alloc(N*sizeof(int), omp_get_default_device());

array_host = (int *) malloc(N*sizeof(int));
/* initialize array_host */

#pragma omp target is_device_ptr(array_device) map(tofrom: array_host[0:N])
{
  for (int i = 0; i < N; ++i) {
    array_device[i] = i;
    array_host[i] += array_device[i];
  }
}
...
...
has_device_addr and defaultmap clause on target directive

• has_device_addr clause indicates that the list items already have device addresses
  – list item must have a valid device address for the device data environment
  – if not, leads to unspecified behavior

• defaultmap clause is used to change implicitly determined data-mapping and data-sharing attribute rules of variables referenced in the target region
  – Example: defaultmap(tofrom: scalar)
OpenMP Offload: Example using `omp target`

/*C code to offload Matrix Addition Code to Device with map clause using static arrays*/

```c
... int A[N][N], B[N][N], C[N][N];
/* initialize arrays */
#pragma omp target map(to: A, B) map(from: C)
{
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
} // end target
```

/*C code to offload Matrix Addition Code to Device with map clause using dynamic arrays*/

```c
... int *A, *B, *C;
/* allocate arrays of size N and initialize */
#pragma omp target map(to: A[0:N], B[0:N]) map(from: C[0:N])
{
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
} // end target
```

Array Sections
Array Sections in OpenMP

• An array section designates a subset of the elements in an array.

  
  

• Must be a subset of the original array.

• Array sections are allowed on multidimensional arrays.

• Must be integers or integer expressions
  – The length must evaluate to a non-negative integer and must be explicitly specified

• when the size of the array dimension is not known
  – The stride must evaluate to a positive integer, default 1 – lower-bound when absent it defaults to 0.

[[ lower-bound] : length [: stride]]
Target Device Data Persistence

/* C code for multiple offload kernels */

...  
#pragma omp target map(to: A, B) map(from: C)  
  {  
    for (int i = 0; i < N; ++i) {  
      for (int j = 0; j < N; ++j) {  
        C[i][j] = A[i][j] + B[i][j];  
      }  
    }  
  }  
/*  
 Some computation using C (no changes to A, B or C)  
*/  
#pragma omp target map(to: A, B, C) map(from: D)  
  {  
    for (int i = 0; i < N; ++i) {  
      for (int j = 0; j < N; ++j) {  
        D[i][j] = A[i][j] + B[i][j] C[i][j];  
      }  
    }  
  }  
...  

Is this optimal?  

NO  

A and B are unchanged between the two target regions.
### OpenMP Device Data Directives

<table>
<thead>
<tr>
<th>C/C++ API</th>
<th>Fortran API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!$omp target data clause[ [ , ] clause ] ... ]</code></td>
<td><code>$omp target data clause[ [ , ] clause ]</code></td>
<td></td>
</tr>
<tr>
<td><code>!$omp end target data</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#pragma omp target enter data [clause[ , ] clause ] ... ]</code></td>
<td><code>$omp target enter data [clause[ , ] clause ]</code></td>
<td>A standalone directive that specifies that variables are mapped to a device data environment. It does so via a <code>map</code> clause.</td>
</tr>
<tr>
<td><code>!$omp target enter data</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#pragma omp target exit data [clause[ , ] clause ] ... ]</code></td>
<td><code>$omp target exit data [clause[ , ] clause ]</code></td>
<td>A standalone directive that specifies that variables are unmapped from a device data environment via a <code>map</code> clause.</td>
</tr>
<tr>
<td><code>!$omp target exit data</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
/*C code for multiple offload kernels with structured data mapping using target data map*/

#pragma omp target data map(to: A, B)
{
    #pragma omp target map(from: C) //kernel 1
    {
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                C[i][j] = A[i][j] + B[i][j];
            }
        }
    }

    //some computation on host using C (no changes to A, B or C)
    /*
    */

#pragma omp target map(to: C) map(from: D) //kernel 2
{
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            D[i][j] = A[i][j] + B[i][j] C[i][j];
        }
    }
}

} //end target-data
/*C code for multiple offload kernels using target enter/exit data map*/

void foo()
{
#pragma omp target enter data map(to: A, B)
}

void bar()
{
...
#pragma omp target map(to: C)
{
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        D[i][j] = A[i][j] + B[i][j] C[i][j];
    }
}

#pragma omp target exit data map(release: C)
map(from: D)
}

int main()
{
...
    foo();

#pragma omp target map(from: C)
{
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
} //end target
...
    bar();
...
}
target update construct

• Syntax

  C/C++ : #pragma omp target update clause[ [ [,] clause] ... ]

  Fortran: !$omp target update clause[ [ [,] clause] ... ]

• The target update directive makes list items consistent according to the specified data-motion-clauses.
  - update to makes the corresponding list items in the target device data environment consistent with their original list items
  - update from makes the original list item in the host device data environment consistent with their corresponding list items on the target device data environment
/*C code for multiple offload kernels using target data map and target update*/

... 
#pragma omp target data map(to: A, B) map(alloca: C, D) 
{
#pragma omp target
{
   for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
         C[i][j] = A[i][j] + B[i][j];
      }
   }
}
#pragma omp target update from(C) //Updates C device → host

/*
Some changes to A (no changes to B or C)
*/
#pragma omp target update to(A) //Updates A host → device

#pragma omp target map(from: D) 
{
   for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
         D[i][j] = A[i][j] + B[i][j] C[i][j];
      }
   }
}
//end target-data
Declare Target directives

• Applied to procedures and/or variables to ensure that they can be executed or accessed on a device.

• A variable declared in the directive must:
  – have a mappable type
  – have static storage duration

• Two variations:
  – declare target directive (clauses: enter, link, device_type, indirect)
  – begin declare target directive (clauses: device_type, indirect)
    • Must be paired with end declare target directive
/*C code for demonstrating use of declare target*/

#pragma omp begin declare target
int a[N], b[N], c[N];
int i = 0;
#pragma omp end declare target

void update() {
    for (i = 0; i < N; i++) {
        /*update a, b, and c*/
    }
}

#pragma omp declare target to(update)

int main() {
    update();
    #pragma omp target update to(a,b,c)
    #pragma omp target
    {
        update();
    }
    #pragma omp target update from( a, b, c)
## Device Memory Query Routines

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int omp_target_is_present(const void *ptr, int device_num);</code></td>
<td><code>integer(c_int) function omp_target_is_present(ptr, device_num)</code></td>
<td>routine tests whether a host pointer refers to storage that is mapped to a given device.</td>
</tr>
<tr>
<td></td>
<td><code>&amp; bind(c) use, intrinsic :: iso_c_binding, only : c_ptr, c_int</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int omp_target_is_accessible(const void *ptr, size_t size, int device_num);</code></td>
<td><code>integer(c_int) function omp_target_is_accessible( &amp; ptr, size, device_num)</code> bind(c) use, intrinsic :: iso_c_binding, only : c_ptr, c_int`</td>
<td>routine tests whether host memory is accessible from a given device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>void * omp_get_mapped_ptr(...);</code></td>
<td><code>type(c_ptr) function omp_get_mapped_ptr(...)</code></td>
<td>routine returns the device pointer that is associated with a host pointer for a given device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Device Memory Copy Routines

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int omp_target_memcpy_rect(..);</code></td>
<td><code>integer(c_int) function omp_target_memcpy_rect(dst,src,element_size, &amp; ..)</code></td>
<td>copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array.</td>
</tr>
<tr>
<td><code>int omp_target_memcpy_async(..);</code></td>
<td><code>integer(c_int) function omp_target_memcpy_async( ..)</code></td>
<td>routine asynchronously performs a copy between host and device pointers.</td>
</tr>
<tr>
<td><code>int omp_target_memcpy_rect_async(..);</code></td>
<td><code>integer(c_int) function omp_target_memcpy_rect_async( ..)</code></td>
<td>routine asynchronously performs a copy between host and device pointers.</td>
</tr>
<tr>
<td><code>int omp_target_memcpy(void *dst,..);</code></td>
<td><code>integer(c_int) function omp_target_memcpy(dst, src, length, &amp; 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 ..</code></td>
<td>routine copies memory between host and device pointers.</td>
</tr>
</tbody>
</table>
### Other Device Memory Routines

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int omp_target_associate_ptr(...);</code></td>
<td><code>integer(c_int) function omp_target_associate_ptr(...</code></td>
<td>routine maps a device pointer to a host pointer</td>
</tr>
<tr>
<td><code>int omp_target_disassociate_ptr(...);</code></td>
<td><code>integer(c_int) function omp_target_disassociate_ptr(…..</code></td>
<td>routine removes the associated pointer for a given device from a host pointer.</td>
</tr>
</tbody>
</table>