OpenMP 4.5 target

• Wednesday, June 28th, 2017

Presenters:
Tom Scogland
Oscar Hernandez

Credits for some of the material
IWOMP 2016 tutorial – James Beyer, Bronis de Supinski
OpenMP 4.5 Relevant Accelerator Features – Alexandre Eichenberger
OpenMP 4.5 Seminar – Tom Scogland
What’s new in OpenMP 4.0/4.5

• Directives
  – Target regions (to support accelerators)
    • structure and unstructured target data regions
    • Asynchronous execution (nowait) and data dependences (depend)
  – SIMD (to support SIMD parallelism)
  – New tasking features
    • taskloops, groups, dep, priorities
    • Cancellation
  – Thread affinity
    • Per parallel region (including nested parallelism)
  – Do across
    • Ordered (do across)

• Runtime APIs
  – Target regions, data mapping APIs

• Environment Variables
  – Affinity:
    • OpenMP Places (OMP_PLACES)
      – Hardware abstraction
    • Thread bindings (OMP_PROC_BIND)
      – Controls the mapping of threads to places
  – Target
    • Default accelerator type
OpenMP 4.0/4.5 – Accelerator model

- OpenMP 4.0/4.5 supports heterogeneous systems (accelerators/devices)
- Accelerator model
  - One host device and
  - One or more target devices

Host Device (CPU Multicore)  | Xeon Phi(s) – (Accelerator and self-hosted)  | GPU(s)
---|---|---
Single device attached

With attached accelerator(s)
OpenMP Target

• Device:
  – An implementation-defined (logical) execution unit (or accelerator)

• Device data environment
  – Storage associated with the device

• The execution model is host-centric (or initial device)
  – Host creates/destroys data environment on device(s)
  – Host maps data to the device(s) data environment
  – Host offloads OpenMP target regions to target device(s)
  – Host updates the data between the host and device(s)
OpenMP 4.5 Device Constructs

• Execute code on a target device
  – **omp target** [clause[.,] clause], …
    structured-block
  – **omp declare target**
    [function-declarations]

• Manage the device data environment
  – **map** ([map-type:] list)
    map-type := alloc | tofrom | to | from | release | delete
  – **omp target data** [clause[.,] clause], …
    structured-block
  – **omp target enter/exit data** [clause[.,] clause], …
  – **omp target update** [clause[.,] clause], …
  – **omp declare target**
    [variable-declarations]

• Parallelism & Workshare for devices
  – **omp teams** [clause[.,] clause], …
    structured-block
  – **omp distribute** [clause[.,] clause], …
    for-loops

• Device Runtime Support
  – void **omp_set_default_device**(int dev_num)
  – int **omp_get_default_device**(void)
  – int **omp_get_num_devices**(void)
  – int **omp_get_team_num**(void)
  – int **omp_is_initial_device**(void)
  – …

• Environment variables
  – **OMP_DEFAULT_DEVICE**
  – **OMP_THREAD_LIMIT**
OpenMP target example

An example of OpenMP 4.5 for accelerators

```c
!$omp target map(to:u) map(from:uold)

!$omp parallel do collapse(2)
  do j=1,m
    do i=1,n
      uold(i,j) = u(i,j)
    enddo
  enddo

!$omp end target
```

Use target construct to:
- Transfer control from the host to the target device
- Map variables to/from the device data environment

Host thread waits until target region completes
- Use nowait for asynchronous execution

initialize device
allocates: u, uold on device data environment
copies in: u

Device initializes
allocate u, uold on device data environment

barrier

Execute target
code

copies out: uold
deallocates: u, uold

Executed on
the device

host thread
OpenMP Target and Data Regions

- The map clauses determine how an original (initial device) variable in a data environment is mapped to a corresponding variable in a device data environment
  - Mapped variable:
    - An original variable in a (host) data environment has a corresponding variable in a device data environment
  - Mapped type:
    - A type that is amenable for mapped variables
    - Bitwise copy-able plus additional restrictions
League and teams of threads

- **League**
  - Set of thread teams created by a teams construct

- **Contention group**
  - Threads of a team in a league and their descendant threads
  - Threads can synchronize in the same contention group

```plaintext
#pragma omp teams
// creates N teams of size 1

#pragma omp target
// creates initial target thread

#pragma omp parallel
// creates M threads within a team
```
teams construct

• The **teams** construct creates a league of thread teams
  – The master threads of all teams execute the team region
  – The number of teams is specified by the **num_teams** clause
  – Each team executes with **thread_limit** threads
  – Threads in different teams cannot synchronize with each other
  – Must be perfectly nested in a target construct
    • No statements or directives between teams and target constructs
  – Only special openmp constructs can be nested inside a teams construct
    • distribute
    • parallel
    • parallel for
    • parallel sections
distribute construct

• Work-sharing construct for target and teams regions
  – Distribute the iterations of a loop across the master threads of the teams executing the region
  – No implicit barrier at the end of the construct

• dist_schedule(kind[, chunk_size])
  – kind must be static scheduling
  – Chunks are distribute in round-robing fashion with chunk_size
  – Each team receives at least one evenly distributed chunk (if no chunk_size is specified)

```c
#pragma omp target map(tofrom: A)
#pragma omp teams
#pragma omp distribute
for (i=0; i< N; i ++)
    A[i] = ....
```
Writing Portable Device code

```c
#pragma omp target map(tofrom: A)
#pragma omp teams distribute parallel for simd collapse(3) // combined directive

for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        for(k=0; k<N; k++)
            A[i][j][k] = ....
```

- Use OpenMP 4 “Accelerator Model” to target multiple architectures: GPUs, Intel Xeon Phi, and multicore CPUs, etc.
- Make your OpenMP adaptable or using defaults for:
  - # of teams,
  - dist_schedule,
  - thread_limit #,
  - # of threads in parallel regions,
  - parallel for loop schedules,
  - SIMD length

Example:
- Xeon Phi implementation may chose num_teams(1), thread_limit(1) and simdlen(V)
- GPUs implementation may chose num_teams(N), thread_limit(M) and simdlen(V)
OpenMP 4.5 and Beyond -- Webinar part 2
Target updates

Wednesday, March 29th, 2017

Scaling OpenMP via LLVM for Exascale Performance and Portability (SOLVE)

Presenters:
Tom Scogland
Oscar Hernandez
Christopher Earl
OpenMP 4.5 Target Features

- Runtime
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
- Target memory routines
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
- Target memory routines
- Target directives
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
- Target memory routines
- Target directives
  - unstructured mapping (enter/exit data)
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
- Target memory routines
- Target directives
  - unstructured mapping (enter/exit data)
  - nowait
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
- Target memory routines
- Target directives
  - unstructured mapping (enter/exit data)
  - nowait
- Target data sharing
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
- Target memory routines
- Target directives
  - unstructured mapping (enter/exit data)
  - nowait
- Target data sharing
  - Structure subsets
OpenMP 4.5 Target Features

- Runtime
- Pointer interoperability
- Target memory routines
- Target directives
  - unstructured mapping (enter/exit data)
  - nowait
- Target data sharing
  - Structure subsets
  - private/firstprivate
Target data sharing updates
Mapping in OpenMP 4.0

```c
map([map-type: ] list-item[, list-item...])
map-type: alloc | to | from | tofrom
list-item: <variable-name>[array-section]
array-section: [<start>:]<length>
```
OpenMP 4.0 Mapping semantics

- All mappings include a presence check
OpenMP 4.0 Mapping semantics

- All mappings include a presence check
- All un-listed variables are mapped *tofrom*
OpenMP 4.0 Mapping semantics

- All mappings include a presence check
- All un-listed variables are mapped *tofrom*
- Array sections are mapped in two parts:
OpenMP 4.0 Mapping semantics

- All mappings include a presence check
- All un-listed variables are mapped to/from
- Array sections are mapped in two parts:
  - A device scalar to store the address of the device buffer
OpenMP 4.0 Mapping semantics

- All mappings include a presence check
- All un-listed variables are mapped *tofrom*
- Array sections are mapped in two parts:
  - A device scalar to store the address of the device buffer
  - A device buffer to store the data
OpenMP 4.0 Mapping semantics

- All mappings include a presence check
- All un-listed variables are mapped to/from
- Array sections are mapped in two parts:
  - A device scalar to store the address of the device buffer
  - A device buffer to store the data
- Presence checks operate on the address of the list-item, never the value
OpenMP 4.0 Mapping semantics

- All mappings include a presence check
- All un-listed variables are mapped *tofrom*
- Array sections are mapped in two parts:
  - A device scalar to store the address of the device buffer
  - A device buffer to store the data
- Presence checks operate on the address of the list-item, never the value
- A pointer referenced in a region but not mapped may be an error or mapped *tofrom* as a scalar
Presence Examples: local

```c
void omp4_foo(double *arr, int len, double arg){
    #pragma omp target data map(tofrom: arr)
    #pragma omp target teams distribute parallel for
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}
```
Presence Examples: local

```c
void omp4_foo(double *arr, int len, double arg){
    #pragma omp target data map(tofrom: arr)
    #pragma omp target teams distribute parallel for
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}
```
Mapping a pointer gives you a copy of the pointer value, and almost certainly a segfault
Presence Examples: call

```c
void omp4_target_foo(double *arr, int len, double arg){
    #pragma omp target teams distribute parallel for
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}
void omp4_foo(double *arr, int len, double arg){
    #pragma omp target data map(tofrom: arr)
    omp4_target_foo(arr, len, arg);
}
```
Presence Examples: call

```c
void omp4_target_foo(double *arr, int len, double arg) {
    #pragma omp target teams distribute parallel for
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}

void omp4_foo(double *arr, int len, double arg) {
    #pragma omp target data map(tofrom: arr)
    omp4_target_foo(arr, len, arg);
}
```
Presence checks are by address of the list-item, the array is not found and may be remapped
Presence Examples: call, try 2

```c
void omp4_target_foo(double *arr, int len, double arg)
{
    #pragma omp target teams distribute parallel for \
    map(tofrom: arr[0:len])
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}

void omp4_foo(double *arr, int len, double arg)
{
    #pragma omp target data map(tofrom: arr)
    omp4_target_foo(arr, len, arg);
}
```
Presence Examples: call, try 2

```c
void omp4_target_foo(double *arr, int len, double arg) {
    #pragma omp target teams distribute parallel for \
    map(tofrom: arr[0:len])
    for (int i = 0; i < len; ++i)
        arr[i] *= arg;
}
void omp4_foo(double *arr, int len, double arg) {
    #pragma omp target data map(tofrom: arr)
    omp4_target_foo(arr, len, arg);
}
```
# Presence Examples: unstructured map

```c
void map_arr(double *arr, int len) {
    #pragma omp target enter data map(to: arr)
}

void omp4_target_foo(double *arr, int len, double arg) {
    #pragma omp target teams distribute parallel for \
        map(tofrom: arr[0:len])
    for (int i = 0; i < len; ++i)
        arr[i] *= arg;
}

void omp4_foo(double *arr, int len, double arg) {
    map_arr(arr, len);
    omp4_target_foo(arr, len, arg);
}
```
Presence Examples: unstructured map

```c
void map_arr(double *arr, int len) {
    #pragma omp target enter data map(to: arr)
}

void omp4_target_foo(double *arr, int len, double arg) {
    #pragma omp target teams distribute parallel for \
    map(tofrom: arr[0:len])
    for (int i = 0; i < len; ++i)
        arr[i] *= arg;
}

void omp4_foo(double *arr, int len, double arg) {
    map_arr(arr, len);
    omp4_target_foo(arr, len, arg);
}
```
Why didn't that Work?

- target enter data maps arr by the address of the local variable
Why didn't that Work?

- target enter data maps arr by the address of the local variable
- map fails to find it, because it looks for the address of a non-existant stack variable
Why didn't that Work?

- target enter data maps arr by the address of the local variable
- map fails to find it, because it looks for the address of a non-existant stack variable
- map maps it again
Why didn't that Work?

- target enter data maps arr by the address of the local variable
- map fails to find it, because it looks for the address of a non-existent stack variable
- map maps it again
- The mapping is *lost!*
In Fortran?

```fortran
!$omp target data map(to:f) map(tofrom:u) map(alloc:uold)
   do while (k.le.maxit .and. (error.gt.tol .or. k .eq. 1))
!$omp target teams distribute parallel do reduction(+:error)
   do j = 2,m-1
!$omp simd private(resid) reduction(+:error)
      do i = 2,n-1
         resid = (ax*(uold(i-1,j) + uold(i+1,j))
         & + ay*(uold(i,j-1) + uold(i,j+1))
         & - f(i,j)) * brecip + uold(i,j)
         u(i,j) = uold(i,j) - omega * resid
      end do end do end do
!$omp end target teams distribute parallel do
```
Refined Mapping in OpenMP 4.5
Type-based Implicit Mappings

- Scalars: firstprivate by default, can be mapped explicitly to change
Type-based Implicit Mappings

- Scalars: firstprivate by default, can be mapped explicitly to change
- Pointers/arrays: default present, looked up by the value of the pointer
Type-based Implicit Mappings

- Scalars: firstprivate by default, can be mapped explicitly to change
- Pointers/arrays: default present, looked up by the *value* of the pointer
- Other: still mapped to/from
Data Only Array Sections

- When mapping a array, presence checks are performed on the value of the pointer
Data Only Array Sections

- When mapping a array, presence checks are performed on the value of the pointer
- The scalar holding the pointer is firstprivate, only the data it points to is mapped
Data Only Array Sections

- When mapping a array, presence checks are performed on the value of the pointer
- The scalar holding the pointer is firstprivate, only the data it points to is mapped
- No more dead stack variables in the presence table!
Data Only Array Sections

- When mapping a array, presence checks are performed on the value of the pointer
- The scalar holding the pointer is firstprivate, only the data it points to is mapped
- No more dead stack variables in the presence table!
- (unless the user asks for it...)
Presence Examples: unstructured map revisited

```c
void map_arr(double *arr, int len) {
    #pragma omp target enter data map(to: arr)
}

void omp4_target_foo(double *arr, int len, double arg) {
    #pragma omp target teams distribute parallel for \
    map(tofrom: arr[0:len])
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}

void omp4_foo(double *arr, int len, double arg) {
    map_arr(arr, len);
    omp4_target_foo(arr, len, arg);
}
```
Presence Examples: unstructured map revisited

```c
void map_arr(double *arr, int len) {
    #pragma omp target enter data map(to: arr)
}
void omp4_target_foo(double *arr, int len, double arg) {
    #pragma omp target teams distribute parallel for \map(tofrom: arr[0:len])
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}
void omp4_foo(double *arr, int len, double arg) {
    map_arr(arr, len);
    omp4_target_foo(arr, len, arg);
}
```
Presence Examples: call revisited

```c
void omp4_target_foo(double *arr, int len, double arg){
    #pragma omp target teams distribute parallel for \
    map(tofrom: arr[0:len])
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}

void omp4_foo(double *arr, int len, double arg){
    #pragma omp target data map(tofrom: arr)
    omp4_target_foo(arr, len, arg);
}
```
Presence Examples: call revisited

```c
void omp4_target_foo(double *arr, int len, double arg) {
    #pragma omp target teams distribute parallel for \
    map(tofrom: arr[0:len])
    for (int i=0; i < len; ++i)
        arr[i] *= arg;
}

void omp4_foo(double *arr, int len, double arg) {
    #pragma omp target data map(tofrom: arr)
    omp4_target_foo(arr, len, arg);
}
```
Sub-mapping for structures
Example: Sub-mapping

```c
struct a {
    float huge_unnecessary_array[10][10][10][10];
    int len;
    double *buf;
};
...
#pragma omp target map(to: a.len, a.buf[0:a.len])
```
Target directives
Unstructured mapping

• Begin an unstructured data scope:

```c
#pragma omp target enter data [to()] [alloc()]
```
Unstructured mapping

- Begin an unstructured data scope:
  
  ```
  #pragma omp target enter data [to()] [alloc()]
  ```

- End an unstructured data scope:

  ```
  #pragma omp target exit data [from()] [release()] [delete()]
  ```
Example: Unstructured mapping

class OMPVector {
    std::vector<float> vec;
    OMPVector(size_t size) : vec(size) {
        float * ptr = vec.data();
        #pragma omp enter data alloc(ptr[0:size])
    }
    ~OMPVector() {
        float * ptr = vec.data();
        #pragma omp exit data delete(ptr)
    }
    ...
}
The nowait clause
All target regions are now tasks!
Changes in the target execution model

• Without nowait, nothing changes...
Changes in the target execution model

- Without nowait, nothing changes...
- Except that it will wait on dependencies
Changes in the target execution model

- Without nowait, nothing changes...
- Except that it will wait on dependencies
- With nowait
Changes in the target execution model

- Without nowait, nothing changes...
- Except that it will wait on dependencies
- With nowait
  - The region may run asynchronously
Changes in the target execution model

- Without nowait, nothing changes...
- Except that it will wait on dependencies
- With nowait
  - The region may run asynchronously
  - It can host dependencies
Changes in the target execution model

- Without nowait, nothing changes...
- Except that it will wait on dependencies
- With nowait
  o The region may run asynchronously
  o It can host dependencies
  o Copies and computation can proceed in parallel!
Example: Pipelining

```c
void tiled_computation (tile_t ** tiles, size_t count) {
    for (int i=0; i < count; ++i) {
        tile_t * tile = tiles[i];
        #pragma omp target enter data map(to:tile[0:1]) depend(inout: tile[0])
        #pragma omp target nowait depend(inout: tile[0])
        process_tile(tile);
        #pragma omp target exit data map(from:tile[0:1]) depend(inout: tile[0])
    }
}
```
Example: Pipelining in Fortran

```fortran
!$OMP TARGET TEAMS DISTRIBUTED DEPEND(OUT:df1) NOWAIT
<compute on df1>
!$OMP END TARGET TEAMS DISTRIBUTED

!$OMP TARGET TEAMS DISTRIBUTED DEPEND(IN:df1) DEPEND(OUT:atmpx1) NOWAIT
<compute on atmpx1>
!$OMP END TARGET TEAMS DISTRIBUTED

!$OMP TARGET UPDATE FROM(atmpx1) DEPEND(INOUT:atmpx1) NOWAIT

!<do things for other coordinate directions>

!$OMP TASK DEPEND(IN:atmpx1)
IF (rankCD .EQ. 0) WRITE(*,*) 'FINISHED WAITING FOR ATMPX1'
!$OMP END TASK
```
What about a host task?
Example: Pipelining and host

```c
void tiled_computation (tile ** tiles, size_t count) {
    for (int i = 0; i < count; ++i) {
        tile * tile = tiles[i];
        #pragma omp target enter data to(tile[0:1]) depend(tile)
        #pragma omp target nowait depend(tile)
        process_tile(tile);
        #pragma omp target exit data from(tile[0:1]) depend(tile)
        #pragma omp task depend(tile)
        {
            post_process(tile);
        }
    }
}
```
Questions?