Device Extensions in OpenMP* 4.0

This tutorial aims to familiarize you with what's new in OpenMP* 4.0. The OpenMP* 4.0 specification introduced support for accelerators, SIMD constructs, error handling, thread affinity, and tasking extensions. OpenMP* is a building block for high-performance computing (HPC), for technical and scientific computing, and for other domains.

This lab focuses in particular on the offload extensions that target for example Intel® Xeon Phi™ coprocessors. This material uses the N-body problem to illustrate the effect of the offload extensions. For the matter of simplicity and to stay on topic, the presented code is not meant to be sophisticated as for example using an acceleration data structure i.e., the problem's complexity is straight forward $O(N^2)$.

Background: an ***N*-body simulation** is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity. In cosmology, they are used to study processes of non-linear structure formation such as the process of forming galaxy filaments and galaxy halos from dark matter in physical cosmology. Direct *N*-body simulations are used to study the dynamical evolution of star clusters.

Build and try the Application

Enter the lab directory where you will find the source files `mxv.cpp` along with a `Makefile`. The default target system is Xeon. If you have an Intel® Xeon Phi™ coprocessor available you may prefer to run it on the coprocessor or both; please adjust the Makefile accordingly. Build and run the application:

```
./make_n_run[_mic | host].sh
```

N-Body C++ Code Sample

Decide what is beneficial to be offloaded to a highly parallel coprocessor. One usually starts with a basic copy-in / copy-out strategy, and optimizes it further by pipelining compute and transfer. The ratio between the total amount of computation and data transfer determines the degree to what asynchronous offloads can hide the time needed for data transfers.

1. Target the function Compute_Acceleration() to be offloaded by actually offloading the body of the function Perform_NBody(). Use the following pragma:
   `#pragma omp target`
   or if you want to specify the target device use:
   `#pragma omp target device(n)`
2. Compile the program (offload is enabled by default), and resolve the issues pointed out by the compiler i.e., identify any routines that are called from inside of the offloaded region. This tells the compiler to generate code versions for the offload-target architecture. Note, that this can be achieved marking an entire section, or on a per-translation unit basis, etc. Use:

#pragma omp declare target
Declare the functions Compute_Acceleration() and Execute() to be offload targets.

3. Repeat the previous step for the global variables:
epsilon_sqr
[…]
nthreads
Use the following pair of pragmas to mark the entire code section:
#pragma omp declare target
#pragma omp end declare target
4. Extend the previously employed offload pragma (**#pragma omp target**) by mentioning the arrays (or array sections) that need to be transferred:

```
#  pragma omp target device(0) \
    map(to:Mass[0:global_number_of_bodies]) \
    map(to:Position0[0:global_number_of_bodies*3])  \
    map(to:Velocity0[0:global_number_of_bodies*3])  \
    map(Acceleration0[0:global_number_of_bodies*3]) \
    map(to:Position1[0:global_number_of_bodies*3])  \
    map(to:Velocity1[0:global_number_of_bodies*3])  \
    map(Acceleration1[0:global_number_of_bodies*3]) \
    map(to:Position0_X[0:global_number_of_bodies])  \
    map(to:Position0_Y[0:global_number_of_bodies])  \
    map(to:Position0_Z[0:global_number_of_bodies])  \
    map(to:Position1_X[0:global_number_of_bodies])  \
    map(to:Position1_Y[0:global_number_of_bodies])  \
    map(to:Position1_Z[0:global_number_of_bodies])
```

Arrays as well as sub-arrays can be transferred using the array section notation. Any dynamically allocated data need to be specified in the fore mentioned manner. However, global variables (even when scalar) also need to be transferred depending on where the current state belongs to. Prior to the pragma omp target, one may use the following pragma in order to transfer the global state:
#pragma omp target update device(0) to(nthreads, […])
and vice versa to update the global state on the host:
#pragma omp target update device(0) from(native_time)

5. Last but not least, spot the omp_set_num_threads(nthreads) call and wrap it into a section using curly braces. Offload this section using **#pragma omp target**. Also, transfer the most recent state of the **nthreads** variable to the device by repeating what you learned in the previous step.
6. Compile and run the program using the **make_n_run.sh** script. Eventually try the other scripts as well, and maybe have a look into **make_n_run_host.sh**. Notice how the offload (enabled by default) is turned off using a compiler option. Record the execution times in the table below!

	Host	Offload	Native
Run time:			

N-Body Fortran Code Sample

Decide what is beneficial to be offloaded to a highly parallel coprocessor. One usually starts with a basic copy-in / copy-out strategy, and optimizes it further by pipelining compute and transfer. The ratio between the total amount of computation and data transfer determines the degree to what asynchronous offloads can hide the time needed for data transfers.

1. Target the function Compute_Acceleration() to be offloaded by actually offloading the body of the function Perform_NBody(). Use the following directives:
   ```
   !$omp target
   !$omp end target
   ```
   Cover the entire executable part of the function Perform_NBody().
2. Compile the program (offload is enabled by default), and resolve the issues pointed out by the compiler i.e., identify all variables that are used in the offload section:
   ```
   !$omp declare target(Position, Acceleration)
   FPTYPE, allocatable :: Position(:), Acceleration(:)
   !$omp declare target(Mass)
   FPTYPE, allocatable :: Mass(:)
   […]
   ```
3. Map the array data that is used in the offload region by extending the previously employed offload pragma (**`!$omp target`**), and mention the arrays (or array sections) that need to be transferred:
   ```
   !$omp target &
   !$omp& map(to:Mass(1:number_of_bodies)) &
   !$omp& map(tofrom:Acceleration(1:3*number_of_bodies)) &
   !$omp& map(to:Position_X(1:number_of_bodies)) &
   !$omp& map(to:Position_Y(1:number_of_bodies)) &
   !$omp& map(to:Position_Z(1:number_of_bodies))
   ```
   Arrays as well as sub-arrays can be transferred using the array section notation. Any dynamically allocated data ("allocatable") need to be specified in the fore mentioned manner.
4. Compile and run the program using the script (and option) "**`make_n_run.sh –f`**". Why does the "validation" of the program suggest it is not working as expected? Check whether an actual offload along with transfers appeared for the device. Try:
   ```
   export OFFLOAD_REPORT=<level: 1…3>
   ```
5. Note that global variables (even when scalar) also need to be transferred depending on where the current state belongs to. Prior to the directive **`!$omp target`**, one may use the following directive in order to transfer the global state:
   ```
   !$omp target update to(number_of_bodies)
   ```
6. Compile and run the program by using "**`make_n_run.sh –f`**" (offload). Try the other scripts as well (host and native, and record the execution times!

	Host	Offload	Native
Run time:			