Device Extensions in OpenMP* 4.0

This lab focuses on the offload extensions of OpenMP* 4.0 which target for example Intel® Xeon Phi™ coprocessors. We follow a simple process:

1. Identify hotspots that can benefit from offloading to the coprocessor.
2. Create offload versions of the hotspots and related variables.
3. Optimize data transfers between the host and the coprocessor.
4. Further optimization.

This process can be applied to any code, and it is not specific to OpenMP* 4.0 – the conversion could also make use of Intel® LEO and to a limited extent other techniques, such as pyMIC, as well.

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.

Setup

Building the Application

Begin by loading the Intel Compiler module:

```
module load intel/15.0.2
```

Decide whether you would like to work with C\C++ or Fortran. The techniques covered in this lab are applicable to both, so you should choose whichever language is most familiar. Begin by copying the **`Makefile_<c|ftn>`** of the language of choice to **`Makefile`**.

For C\C++

```
cp Makefile_c Makefile
```

For Fortran

```
cp Makefile_ftn Makefile
```

Then, for both C\C++ and Fortran, you can then build the **`nbody-offload`** application by

```
make
```


or explicitly

```
make nbody-offload
```

The optimization report will be redirected to the file **nbody.optrpt**. Please notice that the structure of the report is dependent on the compiler version used!

In order to build application without offloading support (by using **–no-openmp-offload**) for an Intel® Xeon Phi™ coprocessor, type:

```
make nbody
```


Configuring and Running the Application

Host with coprocessor offload

Coprocessor execution environment is controlled similarly to that of the host. To separate the environment variables of the host and the coprocessor, environment variable MIC_ENV_PREFIX can be used. Set the environment variable as below.

```
export MIC_ENV_PREFIX=MIC
```

Now environment variable MIC_<variables> will set <variable> on the coprocessor side. For instance, using

```
export MIC_OMP_NUM_THREADS=60
```

sets the number of threads to be used in the offload regions to 60.

Before running the offload application, you can set the OpenMP thread affinity as below. The optional **verbose** flag will show the binding of threads to logical processors:

```
export KMP_AFFINITY=compact,granularity=fine[,verbose]
export MIC_KMP_AFFINITY=compact,granularity=fine[,verbose]
```

or, alternatively, by using the OpenMP 4.0 thread affinity syntax

```
export OMP_PLACES=threads
export OMP_PROC_BIND=close
[export KMP_AFFINITY=verbose]
export MIC_OMP_PLACES=threads
export MIC_OMP_PROC_BIND=close
[export MIC_KMP_AFFINITY=verbose]
```

Then, to execute the offload application, run:

```
./nbody-offload 262144
```

Run the application now, and record the checksum and run time:

Checksum:	
Run time:	

To get additional information on offload kernels executed and data transfers performed, you can use the **OFFLOAD_REPORT=<n>** environment variable as follows.

```
export OFFLOAD_REPORT=2
```

The level of detail **n** in the report can be set to values from 1 to 3 where higher values indicate more details:

- **n=1**: Basic report with host/offload computation time in seconds.

- **n=2**: Detailed report including the amount of data transferred.
- **n=3**: Very detailed report including information about individual data transfers.

Set the environment variable to a value of your choosing and re-run the application. Was any offloading performed at all?

Identifying offload hotspots

Since nothing has been done so far to the application to enable offloading, the whole application executes normally on the host. Good offload candidates can by using a profiling tool such as Intel® VTune™ Amplifier or by general knowledge of the application hotspots. A typical offload candidate has a high flops/transferred data ratio in order to offset the time spent transferring data to the coprocessor. For instance, some typical indicators are

- High arithmetic intensity
- Some (near) static data components which can reside on the coprocessor side for the duration of the whole computation or can be updated infrequently

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.

The actual analysis of the application for identifying the offload hotspots will be postponed until the Tools session has taken place. The most time consuming routine of the application is **`Perform_NBody()`** (C\C++) or **`perform_nbody()`** (Fortran). This is the routine where all the computation takes place and which will be chosen as an offload candidate.

Enabling offloading to a coprocessor

Add constructs to offload the body of the function **`Perform_NBody()`**. For offloading, use the OpenMP **`target`** construct.

For C\C++:

```
#pragma omp target [device(n)] map(…)
```

For Fortran:

```
!$omp target [device(n)] map(…)
!$omp end target
```

Optional **`device(n)`** device clause specifies the coprocessor to offload to.

Use the **`map`** clause to specify the range of data to be moved to the coprocessor. Make sure to move **all** the data needed in the arrays **`Position_X`**, **`Position_Y`**, **`Position_Z`**, **`Mass`** and **`Acceleration`**. Do not yet attempt to minimize the amount of data transferred. Since the arrays in question are global variables, you should first locate where they are defined and then declare them also to reside on the coprocessor. This can be achieved by using **`declare target`** –directive as follows (on the declarations section).

For C\C++:

```
#pragma omp declare target
FPTYPE *Acceleration;
FPTYPE *Position_X, *Position_Y, *Position_Z;
FPTYPE *Mass;
#pragma omp end declare target
```

For Fortran:

```
!$omp declare target(Position_X, Position_Y, Position_Z, Mass, &
!$omp                Acceleration)
```

The whole target clause should read as follows.
For C\C++:

```
#pragma omp target map(Position_X[0:num_bodies], \
          Position_Y[0:num_bodies], Position_Z[0:num_bodies], \
          Mass[0:num_bodies], Acceleration[0:3*num_bodies])
```

For Fortran:

```
!$omp target map(Position_X(1:num_bodies), &
!$omp        Position_Y(1:num_bodies), Position_Z(1:num_bodies), &
!$omp        Mass(1:num_bodies), Acceleration(1:3*num_bodies))
```

Compile and run the program and record the execution times in the table below. Make sure that the checksum of the offload run matches that of the host run!

	Host	Offload
Run time:		

Optimizing data transfer directions

When offload computations are performed, minimization of the data transfer is often one of the most important optimizations. OpenMP device model offers several ways to minimize the data transfers. We are now going to focus on optimizing the directions of data transfers between the host and the coprocessor.

Enable detailed offload report with environment variable as follows

```
export OFFLOAD_REPORT=3
```

Run the offload program and record the amount of data transferred between the host and to coprocessor in the table at the very end of this section.

At the end of the computation, the host often needs only the results of the computation. In this case, the final results are stored in the **Acceleration** array. Other array are just input data, i.e., it is sufficient to ensure that those arrays are sent to the coprocessor before the actual computation begins.

Modify the map-clause in the target such that arrays **Position_X**, **Position_Y**, **Position_Z** and **Mass** are transferred **to** the coprocessor and **Acceleration** array **tofrom** the coprocessor. The whole target construct should read as follows.

For C\C++:

```
#pragma omp target map(to:Position_X[0:num_bodies], \
          Position_Y[0:num_bodies], Position_Z[0:num_bodies], \
          Mass[0:num_bodies]) map(tofrom:Acceleration[0:3*num_bodies])
```

For Fortran:

```
!$omp target map(to:Position_X(1:num_bodies), &
!$omp       Position_Y(1:num_bodies), Position_Z(1:num_bodies), &
!$omp       Mass(1:num_bodies)) map(tofrom:Acceleration(1:3*num_bodies))
```

Now compile and run the program and record the execution times in the table below. Make sure that the checksum of the offload run matches that of the host run!

	Host	Offload
Run time:		

Also record the amount of data transferred between the host and the coprocessor to the table below. Compare the amount of the data transferred between the versions with/without data transfer optimizations.

	Original	Optimized
Data to (bytes):		
Data from (bytes):		

Minimizing data transfers

Other common approach for minimizing unnecessary data transfers between the host and the coprocessor is to store some of the transferred data on the coprocessor. Any stored data, assuming it is up to date, can then be re-used in later computations and any data transfer costs related to that data amortized.

Now let us assume that the computation will be performed multiple times for the same initial positions of the data. This is simulated in the skeleton code with a warmup round before the timed computations in the main routine (in fact, the warmup round will also initialize the coprocessors offload environment, create offload threads for later use etc.). Enclosing the calls to **`Perform_NBody()`** -routine in a **`target data`** construct will enable us to avoid any unnecessary data transfers between the host and the coprocessor.

Add a **`target data`** construct in the main routine, after call to **`Initialize()`** and before the first call to the **`Perform_NBody()`** to create a data environment on the coprocessor. The whole **`target data`** construct should read as follows.

For C\C++:

```
#pragma omp target data map(to:Position_X[0:number_of_bodies],  \
             Position_Y[0:number_of_bodies],                    \
             Position_Z[0:number_of_bodies],                    \
             Mass[0:number_of_bodies])                          \
  map(tofrom:Acceleration[0:3*number_of_bodies])
```

For Fortran:

```
!$omp target data map(to: Position_X(1:number_of_bodies), &
!$omp   Position_Y(1:number_of_bodies),Position_Z(1:number_of_bodies), &
!$omp   Mass(1:number_of_bodies)) &
!$omp   map(tofrom:Acceleration(1:3*number_of_bodies))
```

After the warmup round has been completed, the host reinitializes the **`Acceleration`** array. To have this modification visible in coprocessor memory, a **`target update`** must be made from the host to the coprocessor before the second call to **`Perform_NBody()`**.

Add a **`target update`** after the re-initialization of the **`Acceleration`** array to the main routine. The whole **`target update`** directive should read as follows.

For C\C++:

```
#pragma omp target update to(Acceleration[0:3*number_of_bodies])
```

For Fortran:

```
!$omp target update to(Acceleration(1:3*number_of_bodies))
```

Compile and run the program and record the execution times in the table below. Make sure that the checksum of the offload run matches that of the host run! Also investigate the

offload report and try to understand how much data was transferred between the host and the coprocessor in each phase of the computation.

	Host	Offload
Run time:		

Offloading entire functions

OpenMP target model allows offloading entire functions to the coprocessor. This allows one to transfer whole portions of the program to be executed on the coprocessor in a straightforward manner. This approach also has the benefit of not involving the host in the computations at all.

As a skeleton for this exercise, you should use the original version of the nbody -simulation. This can be located in the original course material package or in the **`solutions`** –directory as **`nbody_v0_orig.<c|F90>`**.

Our goal will be to transfer a large portion of the main program to run on the coprocessor. To this end, we add a target construct in the main program, after a call to **`Initialize()`** such that the construct closes only after the call to **`Checking()`** has been made. As previously, arrays **`Position_X`**, **`Position_Y`**, **`Position_Z`** and **`Mass`** are transferred **`to`** the coprocessor and **`Acceleration`** array **`tofrom`** the coprocessor. The whole target construct should read as follows.

For C\C++:

```
#pragma omp target map(to:Position_X[0:number_of_bodies],          \
  Position_Y[0:number_of_bodies], Position_Z[0:number_of_bodies], \
  Mass[0:number_of_bodies]) \
  map(tofrom:Acceleration[0:3*number_of_bodies])
```

For Fortran:

```
!$omp target map(to:Position_X(1:number_of_bodies), &
!$omp Position_Y(1:number_of_bodies), Position_Z(1:number_of_bodies), &
!$omp Mass(1:number_of_bodies)) &
!$omp map(tofrom:Acceleration(1:3*number_of_bodies))
```

We continue by declaring some global variables to reside on the coprocessor. This can be achieved by using **`declare target`** –directive as follows (on the declarations section).

For C\C++:

```
#pragma omp declare target
FPTYPE *Acceleration;
FPTYPE *Position_X, *Position_Y, *Position_Z;
FPTYPE *Mass;
int number_of_bodies;
FPTYPE epsilon_sqr = 0.01;
#pragma omp end declare target
```

For Fortran:

```
!$omp declare target(Position_X, Position_Y, Position_Z, Mass, &
!$omp                Acceleration, number_of_bodies, epsilon_sqr)
```


Note that global variables (even when scalar) also need to be updated depending on the coprocessor whenever the host state has changed. In our case, after **number_of_bodies** has been read from the command arguments, its value needs to be updated on the coprocessor. Add a **target update** directive after the value of **number_of_bodies** has been read in the main function as follows:

For C\C++:

```
#pragma omp target update to(number_of_bodies)
```

For Fortran:

```
!$omp target update to(number_of_bodies)
```

Now it remains to add target attributes to the called functions themselves. This is done similarly as with variables, i.e., by adding a declare target to the function definitions.

For C\C++ (in the function declaration or header):

```
#pragma omp declare target
void Perform_NBody() { /* Implementation…*/ }
void Checking(){ /* Implementation…*/ }
#pragma omp end declare target
```

For Fortran (in the function declaration part **and** at call sites for functions not in **MODULE**s)

```
subroutine Perform_NBody()
   !...
   !$omp declare target
   !...
end subroutine Perform_NBody
subroutine Checking()
   !...
   !$omp declare target
   !...
end subroutine Checking
```

Since **Perform_NBody()** and **Checking()** are not declared in a **MODULE**, in the declarations section of the main program, add the following

```
!$omp declare target(Perform_NBody, Checking)
```

Compile and run the program and record the execution times in the table below. Make sure that the checksum of the offload run matches that of the host run! Again also investigate the offload report and try to understand how much data was transferred between the host and the coprocessor in each phase of the computation. In this case, is there a way how you could further reduce the amount of data transferred?

	Host	Offload

Run time:		

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3 and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Trademark Information

BlueMoon, BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, E-GOLD, Flexpipe, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel CoFluent, Intel Core, Intel Inside, Intel Insider, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel vPro, Intel Xeon Phi, Intel XScale, InTru, the InTru logo, the InTru Inside logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Pentium, Pentium Inside, Puma, skoool, the skoool logo, SMARTi, Sound Mark, Stay With It, The Creators Project, The Journey Inside, Thunderbolt, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, X-GOLD, XMM, X-PMU and XPOSYS are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Technical Collateral Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software Source Code Disclaimer

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

