Program Development: Offloading

9/28/2013

An application executing on a CPU can instruct the host to execute a block of its code or one of its functions on another device. The devices are often called accelerators or coprocessors, and the process is called offloading. While most HPC users are familiar with programming for clusters, the use of accelerators (GPUs, MICs, FPGAs, ARMs) as offload compute engines is a new realm for many. In this document we provide details that will guide developers and end users in assessing the utility of offloading work onto coprocessors, and describe how to transition to an offload model. There are four points to remember when discussing computations on the Stampede host (E5) CPUs and (Intel Xeon Phi, or simply MIC) coprocessors:

- The instruction sets and architectures of the host E5 and MIC coprocessor are quite similar, but are not identical. (Expect differences in performance.)
- The host processors and the MIC coprocessor each have their own memory subsystem. They are effectively separate SMP systems.
- Application data used on the CPU and coprocessor must be synchronized. Hence there will be data transfers between the CPU and coprocessor memories.
- On Stampede, login nodes do not have coprocessors. Use the "interactive batch" `idev` command to access development compute nodes for development work.

Use the `idev` command below to access a development compute node. The `idev` command returns a compute-node name as a prompt when connected to a compute node. We will use the "c559-001$" prompt to designate commands on a compute node throughout this document.

```
login1$  idev
...
c559-001$
```

There are two offloading models, Automatic Offloading (AO) and Compiler Assisted Offloading (CAO).

Automatic Offload (AO)

Automatic Offloading (AO) is a way to invoke offloading of compute-bound MKL routines, listed in Table 1. This makes sense when these routines use a significant amount of time in your application. AO will automatically determine the fastest mode to use for executing the routine: only on the CPU, solely on the MIC, or combined CPU and MIC execution. On Stampede the Automatic Offload (AO) versions of these routines are included in all applications that load these routines from the MKL library: just include the new shortened MKL option on the compiler command (-mkl) and the AO routines become available in the application.

Type	**Routine**
Level-3 BLAS	xGEMM, xTRSM, STRMM
LAPACK 3 amigos	LU, QR, Cholesky
See MKL for latest additions	

Table 1. Automatic Offload MKL enabled functions.

In the default mode the AO routines just execute on the host (E5 cores), as you would expect. When the `MKL_MIC_ENABLE` environment variable is set to "1" the automatic feature is turned on, and during the application execution a run-time interface determines the best way to run a routine (only on the E5 processors, only on the phi coprocessor, or a division of the work between both). The work fraction for the MIC can be set manually through the `MKL_HOST_WORKDIVISION and KL_MIC_WORKDIVISION` environment variables. MPI applications that use multiple tasks per node will need to adjust the workload division for sharing the coprocessor among all of the tasks.

The following commands show how to compile and run an AO executable.

```
login1$  ifort  -mkl -xhost -O2 app_has_MKLdgemm.f90
...
c559-001$ export MKL_MIC_ENABLE=1
c599-001$ ./a.out
```

Compiler Assisted Offload (CAO)

Developers can explicitly direct a code block to be executed on the MIC in the Fortran and C/C++ languages. The block can be as short as a single routine call (function or subroutine) or as large as a section of code, with or without OpenMP directives. Programming details for offloading can be found in the Intel Compiler XE 13.0 User and Reference Guides:

Fortran: Key Features→Intel MIC Architecture→Programming for Intel MIC Architecture
C/C++: Key Features→Intel MIC Architecture→Programming for Intel MIC Architecture

Code Example 1 illustrates an offload in pseudo code. Directives (explained below) are used to specify that a code block can be offloaded. During compilation the compiler makes a host and coprocessor binary version of an offload code block (the code block is sometimes called an offload region). When the host execution encounters the offload region the runtime performs several offload operations:

- detection of a target Phi (MIC) coprocessor,
- allocation of memory space on the coprocessor,
- data transfer from the host to the coprocessor,
- execution of the coprocessor binary on the MIC,
- transfer of data from the MIC back to the host, and
- deallocation of coprocessor data.

If the runtime does not detect a MIC “target” or it is not available at the time of the offload, the host-binary of the offload code block may be executed. Likewise, when an offload fails, the host-binary may be executed. (Mandatory offloading is possible.)

Data that is used by the CPU and the MIC is managed and synchronized between the two memories, as described in the above offload steps. The offload model is suitable when the data to be exchanged between the host and the MIC consists of scalars, arrays, and Fortran derived types and C/C++ structures that can be copied using a simple memcpy. This characteristic is often described as being flat or byte-wise copyable. The data to be transferred at the offload point may not need to be declared or allocated in any special way on offload directives; although “pointer data” (arrays pointed to by a pointer) need size information. Also data specifiers in the directives can be used to manage coprocessor storage and control transfers.

```
program
 ...
<offload directive, begin>
   code block              with/without OpenMP,
   or                      Cilk, Pthreads, etc.
   function/routine call
<offload region end>
 ...
program end
```

Code Example 1. Pseudo code for offload.

The most important consideration in offloading is whether there is any performance gain in offloading or not. There are two factors that a developer must consider: How much time will it take to transfer data back and forth between the host and the MIC, and how fast is the computation on the coprocessor. For simple blocks of work, a developer should attempt to determine these values from the transfer characteristics (we will include more details on this later), and an estimation of the computation time of the offload code. For more complex codes, it may only be possible to estimate whether it is feasible to consider offloading. On the Stampede system the allocations are based solely on the use of the 16 E5 cores. Adaptation of applications to use the Phi coprocessors provides additional "free" compute power and faster turnaround, and a jump-start into the many-core technologies that will boost scientific computing in this decade.

It isn't necessary to modify the code to be offloaded. Code can be offloaded by simply by placing an offload directive before (and after for Fortran) a block of code. The simplest form of the offload directive consists of a C/C++ pragma (#pragma offload) or Fortran comment (!dir$ offload) directive with a target specifier containing a "mic" (or "MIC") argument and an optional device number after a colon (:device_number). The offload syntaxes for a code block and a single-statement routine call are:

```
#pragma offload     target(mic:dev_id)            C/C++ code block, :dev_id is optional
{...
}

#pragma offload     target(mic:dev_id)            C/C++ function (single statement)
foo(...);

---------------------------------------------------------------------------------------------

!dir$ offload begin target(mic:dev_id)            Fortran code block, :dev_id is optional
...
!dir$ end offload

!dir$ offload       target(mic:dev_id)            Fortran routine (single statement)
call foo(...)                                     No end offload required
```

When the target argument is `mic` (without a device number) the compiler will include instructions to execute the code block or routine on the MIC if it is available; if the MIC is not available, the directive is effectively ignored and the CPU version of the offload code is executed on the CPU. Also, if the coprocessor execution fails, the runtime will run the CPU version. If a device number is specified (0 for a single coprocessor on a node, 0 and 1 for dual-mic nodes), the code is executed on the coprocessor with the specified device number, or the execution exits with an error if that coprocessor is not available. By including mandatory on the offload directive, offloading is forced and an error is reported if the device is not available. mic:-1 will accomplish the same objective as mandatory.

Code Example 2 illustrates a code block offload for a C/C++ program and an equivalent Fortran program. The offload directive directs the compiler to offload the next block of code, a loop parallelized with an OpenMP directive. The offload region is forced to execute on the MIC because the target argument has a device number (`mic:0`). The offload is synchronous; that is, the CPU waits for completion of the offload before resuming its execution after the code block. Before executing the offload region on the MIC, arrays `a`, `b`, and `c` are allocated on, and copied to the coprocessor. Upon completion of the MIC execution the arrays are copied back to the host and freed on the coprocessor; likewise, for the locally used `N` and i variables.

```
int main(){                                 program main
...                                         ...
   #pragma offload target(mic:0)            !dir$ offload begin target(mic:0)
   {
       #pragma parallel for                 !$omp parallel do
       for (i=0; i<N;i+){                   do i = 1,N
          a[i]=sin(b[i])+cos(c[i]);            a(i)=sin(b(i))+cos(c(i))
       }                                    end do
    }                                       !dir$ end offload
...                                         ...
}                                           end program
```

Example Code 2. Offloaded code block with automatic data transfers.

Optimized Data Transfers

If data storage is created in the host program code and used in the offload region it must be copied to the coprocessor at some time for the offloaded execution. The compiler will automatically* copy the data *in*to and *out* from the coprocessor if the data is used within the lexical extent of the offload code block, with some exceptions mentioned below. That is, if the compiler knows about the variables and arrays seen within the offload code block it will automatically perform the copies.

For variables that are only read or only written on the coprocessor an `in` or`out` data specifier can be used to eliminate unnecessary transfers. These specifiers can reduce offload overhead for large arrays. The `inout` specifier copies data to and from the device (the default behavior for automatic* transfers).The syntax of the data specifier (called an offload-parameter by Intel) is:

```
#pragma offload target(mic[:dev#]) data_specifier(identifier_list)//syntax

#pragma offload target(mic:0)   in( b,c )   // Only copy b and c values into MIC
#pragma offload target(mic:0)  out(a    )   // Only return a values out from MIC
#pragma offload target(mic)  inout(d    )   //if mic not available, compute on CPU
```

Multiple `in`, `out` or `inout` specifiers many be used in a directive. In Code Example 3 the data transfers of Code Example 2 have been optimized by specifying the non-default transfers for the large `a`, `b` and `c` arrays.

```
              C code                                        F90 code
int main(){                                   program main
...                                           ...
float a[N], b[N], c[N];                       real :: a(N), b(N), c(N)
...                                           ...

   #pragma offload target(mic:0) \               !dir$ offload begin target(mic:0) &
                   in(a,b) out(c)                                    in(a,b) out(c)
   {
      #pragma parallel for                       !$omp parallel do
      for(i=0;i<N;i++){                          do i=1,N
         a[i]=sin(b[i])+cos(c[i]);                  a(i)=sin(b(i))+cos(c(i))
      }                                          end do
   }                                             !dir$ end offload
...                                           ...
}                                             end program
```

Code Example 3. Offloaded code block with in and out clauses
for optimizing data transfers.

Offload Functions, Globals and C pointers

The C/C++ `_ _attribute_ _ ((target(mic)))` and Fortran `!dir$ attributes offload:mic` directives are used to prefix a function or subroutine declaration for offloading. An often-used alternate form of the C/C++ GNU-type `_ _attribute_ _(( target(mic) ))` directive is the `_ _declspec( target (mic) )` directive.

Global variables that are outside the scope of the offload code (e.g. used in functions and subroutines) must also be "decorated" with the same attributes directive declaration to tell the compiler to include data and provide accessibility in the offloaded code. The compiler option –`offload-attribute-target=mic` will flag every global routine/data_object with a target(mic) attribute. The syntax for the attribute directives is:

```
_ _attribute_ _ (( target(mic) ))   <followed by function/global declaration>    C/C++
_ _declspec      ( target(mic) )    <followed by function/global declaration>

!dir$ attributes offload:mic :: <function/subroutine name or variables>          Fortran
```

Code Example 4 (modified Intel example code) shows the declaration and use of functions and a global variable in an offload code block. (The C/C++ code uses the _ _declspec form of the directive, instead of the of _ _attribute_ _ form.)

C code:

```
 _ _declspec(target(mic))
   int global = 0;

 _ _declspec(target(mic))
   int foo()
   { return ++global; }

main() {
   int i;
   #pragma offload target(mic) inout(global)
   { i = foo(); }

   printf("global:i=%d:%d both=1\n",global,i);
```

Fortran code:

```
  module mydat
    !dir$ attributes offload:mic :: global
    integer :: global = 0
  end module mydat

  !dir$ attributes offload:mic :: foo
  integer function foo
     use mydat
     global = global + 1
     foo = global
  end function foo
```

```
program main
   use mydat
   integer i
   integer,external :: foo
   !dir$ attributes offload:mic :: foo

   !dir$ offload target(mic:0) inout(global)
   i = foo()
   print *, "global:i=",global,i,"(both=1)"

end program main
```

Code Example 4. Decorating functions and globals for offload regions.

The decoration of the function and the global variable are required, even if the function and variable are declared within the same file where the offload code exists. Also, the `inout` data clause on the offload directive is required since the global variable is not seen within the lexical scope of the offload block, and the variable is used and changed in the function. The decoration of the C/C++ function prototype is required and in the Fortran main module it is necessary to describe the external `foo` function as having an offload attribute, even if they have been declared in the same file Note, the single function statement is the offloaded "code block". Single statements don't require enclosing braces(`{…}`) in C/C++, nor the begin-end form of the Fortran directive (`!dir$ offload` ~~**begin**~~ `target(mic) …` ~~`!dir$` **end** `offload`~~).

For C/C++ offloads that use pointers to contiguous elements of data within an offload region, the length of the "pointed to" data must be provided in an `in/out/inout` data specifier as a **modifier**. The modifier is `length(size)`, with an integer expression or a constant for the size argument. Modifiers appear only in data specifiers after a colon, following the variable list. Examples of modifiers are: array length expressions, and the data persistent `alloc_if` and `free_if` expressions described in the next section. The offload syntax and a complete description of the options can be found in the Compiler User Guide (C/C++, Fortran); also, a short description is available from the TACC Offload training module. Below are a few examples of data specifiers with the length modifier:

```
double *a, *b, *c, *d;
...
a=(double *) malloc(N  *sizeof(double));
b=(double *) malloc(N  *sizeof(double));
b=(double *) malloc(N*2*sizeof(double));
c=(double *) malloc(M  *sizeof(double));
...
//syntax #pragma offload target(mic:x) data_specifier(identifier(s): modifier [[,] modifier])
#pragma offload target(mic:0) in( a,b : length(N) )  // pointers a and b have length N
#pragma offload target(mic:0) out(c   : length(N*2)) // pointer c has length N x 2
#pragma offload target(mic)   inout(d : length(M) )  // pointer d has a length of M
```

Persistent Data: alloc_if(), free_if() and offload_transfer

If a loop iteratively executes an offload region, it is probably unnecessary to allocate and deallocate the same storage in each iteration, and if the same data are used throughout, it makes sense to allocate the space on the coprocessor once and copy the data to the coprocessor before the loop, and then deallocate the space after the loop.

By default, at the beginning of an offload, space is allocated on the MIC and then freed at the end of the offload execution. These storage operations can be controlled in offload directives with the `alloc_if(`*arg*`)` and `free_if(`*arg*`)` data modifiers or in an `offload_transfer` directive that is not associated with a computation (code block). The `nocopy` data specifier is often used to manipulate storage with the `alloc_if(`*arg*`)` and `free_if(`*arg*`)` modifiers without transferring data.

The `alloc_if(`*arg*`)` and `free_if(`*arg*`)` modifiers are included in the data specifier in the usual way: they follow the variable list (identifiers) in the data specifier expression, and are separated from the variable list by a colon. The argument of the storage modifier, *`arg`*, determines whether the allocation and deallocation operation is to be performed. It is performed if the *arg* is true (zero, non zero for C/C++, logical true in F90 and C++), and it isn't performed if *arg* is false.

The syntax for the storage operations and examples are:

```
#pragma offload data_specifier(identifier(s): alloc_if(TorF) free_if(TorF) )      //C/C++


#pragma offload    in( a: alloc_if(1) free_if(0) )        //allocate space, don't free at end
{...}
#pragma offload   out( b: alloc_if(0) free_if(1) )        //don't allocate, free at end
{...}
#pragma offload inout( c: alloc_if(0) free_if(0) )      //don't allocate, don't free at end
{...}

#pragma offload_transfer   in( a: alloc_if(1) free_if(0) ) //allocate space, don't free at end
#pragma offload_transfer  out( b: alloc_if(0) free_if(1) ) //don't allocate, free space at end

----------------------------------------------------------------------------------------------

!dir$ offload data_specifier(identifier(s): alloc_if(TorF) free_if(TorF) )       !!Fortran


!dir$ offload begin   in( a: alloc_if(.true. ) free_if(.false.) )
...
!dir$ end offload

!dir$ offload begin  out( b: alloc_if(.false.) free_if(.true. ) )
...
!dir$ end offload

!dir$ offload begin inout( c: alloc_if(.false.) free_if(.false. ) )
...
!dir$ end offload

!dir$ offload_transfer   in( a: alloc_if(.true. ) free_if(.false.) )
!dir$ offload_transfer  out( b: alloc_if(.false.) free_if(.true. ) )
```

The above C/C++ and Fortran directives do identical data management. The first `in` data specifier allocates memory on the MIC for `a` and copies the values of the CPU's `a` to it, and neither copies the data back nor frees the space on the MIC at the end of the offload execution. The `a` data space on the MIC will persist- retain its present values and can be accessed in subsequent offloads. For the first `out` data specifier, neither an allocate for `b` nor a copy of the data is performed; but the data (identified by `b` and which has remained as persistent storage space through a previous modifier in a directive) is copied back from the MIC and its storage is freed. For the `inout` data specifier the CPU data, `c` is copied to the MIC, and the coprocessor data, `c` is copied back at the end of the offload. It is assumed that the space for `c` was previously allocated on the MIC. The coprocessor space is not deallocated at the end of the offload.

The difference between the offload and offload_transfer directives is that the former must have a code block to be executed, while the latter is not associated with any computation, and only manages storage and data transfers. The final two `offload_transfer` directives do the same data operations as the first two `offload` directives in the above examples—they just don't perform any offload computational work.

Table 2 lists the storage operations performed for all possible combinations of operations for the alloc_if and alloc_if modifiers, in a truth table format.

Allocation Operation	**Deallocation (Free) Operation**	**Operations Performed (Use Case)**
alloc_if(true)	free_if(true)	This is the default when no storage operations are specified. Allocate space at beginning, free at end.
alloc_if(true)	free_if(false)	Allocate space, don't free (make space available on device, and retain for future use).
alloc_if(false)	free_if(true)	don't allocate, but free (reuse device storage, but will not need later)
alloc_if(false)	free_if(false)	don't allocate, don't free (reuse device storage, and leave for future use)

Table 2. Truth table for alloc_if and free_if modifier options for the data specifier in/out/inout/nocopy(...) clause.

Code Example 5 shows a use case for persistent data on a coprocessor, using allocation modifiers and offload_transfer directives.

```
program main
integer,parameter  :: N=6000
real*8,allocatable,dimension(:) :: a,b,c
!dir$ attributes offload : mic  :: update

allocate( a(N), b(N), c(N) )
c(:) = 5.0d0

!                  !Make a, b & c persistent on MIC
!dir$ offload_transfer target(mic:0) nocopy( a: alloc_if(.true.) free_if(.false.) )
!dir$ offload_transfer target(mic:0) nocopy( b: alloc_if(.true.) free_if(.false.) )
!dir$ offload_transfer target(mic:0)     in( c: alloc_if(.true.) free_if(.false.) )

do i = 1,50

!dir$ offload  target(mic:0) nocopy(c: alloc_if(.false.) free_if(.false.))  &
                                 in(b: alloc_if(.false.) free_if(.false.))  &
                                out(a: alloc_if(.false.) free_if(.false.))
    call update(i,N,a,b,c)
    b(:) = a(:)

end do
end program

!dir$ attributes offload : mic :: update
subroutine   update(i,n,a,b,c)
real*8, dimension(n) :: a,b,c

   a(:) = b(:)*2.0d0 + c(:)*i

end subroutine
```

Code Example 5. Persistent data example.

In Code Example 5 the stand-alone `offload_transfer` directive is used to create arrays (`alloc_if(.true.)`) and make them persistent by not freeing the space (`free_if(.false.)`). Since the default behavior is to allocate and free the space on any offload data, it really isn't necessary to use the `alloc_if()` modifier. The `nocopy` data specifier specifies that the no data (for arrays `a` and `b`) should be transferred—just create the space on the coprocessor. The `in` modifier for the `c` array specifies that the values in `c` (all 5.0s) should be copied to the space on the coprocessor.

Within the loop, false arguments in the `alloc_if()` and `free_if()` modifiers forces the storage for the arrays to persist across offloads.

The `nocopy` data specifier assures that the same `c` array values are used across offloads (for `c`, data is not copied; and for `a`, `b` and `c` storage is not allocated, and not freed at the end of the offload). The content of the CPU's `a` array is copied to the CPU's `b` array at the end of each iteration. The `in` and `out` data specifiers copy the CPU's `b` array into MIC storage at the beginning of the offload, and copy the `a` array in MIC storage back to the CPU at the end of the offload execution.

Asynchronous Offloading and Data Transfer

The default synchronous behavior of a CPU process (or thread) that encounters an offload directive is to wait for the offload to complete before continuing to execute code after the offload code block. Of course, the associated data transfers are also synchronous. But, the offloads can be made asynchronous, allowing the encountering process to immediately continue execution after the offload region.

The execution of an offload code block can be made asynchronous by including a `signal(tag)` specifier on the offload directive. The offload event is identified by the tag (pointer variable or pointer size value serving as a handle in C/C++, and 8-byte integer or integer in Fortran), and used in the wait specifier in a subsequent `offload` or `offload_wait` directive. Multiple signal tags can be used in a wait specifier; that is, the code can wait on multiple signals. Also, a device id is mandatory in the target specifier. The Fortran tag must be assigned a unique counting integer. A stand-alone `offload_wait` directive can be used to post a wait that is not associated with any offload computation. The syntaxes for these offload components are:

```
#pragma offload      target(mic:dev_id) ... signal(tag)
#pragma offload      target(mic:dev_id) ...   wait(tag_list)
#pragma offload_wait target(mic:dev_id) ...   wait(tag_list)


!dir$   offload      target(mic:dev_id) ... signal(tag)
!dir$   offload      target(mic:dev_id) ...   wait(tag_list)
!dir$   offload_wait target(mic:dev_id) ...   wait(tag_list)

where tag_list is a set of comma separated variables





e.g.
int sig1, sig2
#pragma offload      target(mic:0) signal(&sig1) ...
...
#pragma offload      target(mic:0) signal(&sig2) ...
...

#pragma offload      target(mic:0)   wait(&sig1) ...
...
#pragma offload_wait target(mic:0)   wait(&sig2) ...

-----------------------------------------------------------------------------------------------

integer :: sig1=1, sig2=2
!dir$   offload      target(mic:0) signal(sig1)
...
!dir$   offload      target(mic:0) signal(sig2)
...

!dir$   offload      target(mic:0)   wait(sig1)
...
!dir$   offload_wait target(mic:0)   wait(sig2)
```

This is how previous Intel documentation illustrated the use, and it still works.

The Intel Composer XE release 14 states that the tag is an integer, not integer*8 (KIND=INT_PTR_KIND()) = integer*8 as shown in the composer_xe_2013.2.146 distro leoF11_async.F90 example). The 14 release does not explicitly describe a need to assign a unique value to tags for Fortran code, but shows unique values being set in the manual example: http://software.intel.com/en-us/node/466838#34693A8D-A836-4249-88C9-F67E2483C213.

FYI, the C++ 14.0 Intel User and Reference Guide states:
"*tag* is an expression that is a pointer-size value in the baseline language which serves as a handle on an asynchronous activity, either data transfer or computation".

In Code example 6, a loop iterates over execution of a routine called "work". The function works on sequential elements from `NS` (start) to `NE` (end) in an array of size `N`. The iteration variable, `knt`, determines the operation to be performed on the elements.

Our objective is to split the computation across the CPU and MIC, and execute on the CPU and MIC simultaneously, by offloading the first half of the elements to the MIC while asynchronously executing the work on the last half of the array on the CPU (assume `N` is always even).

```
program main
integer,parameter :: N=10000
integer :: i, knt=1, a(N),NS,NE

do i = 1,N; a(i) = i; end do

  NS=1; NE=N  //start & end

  do while (knt .lt. 101)
    call work(knt,NS,NE, N,a)
    knt=knt+1;
  end do

end program
```

```
#define N 10000
int main(){
int i, knt=0, a[N], NS,NE

for(i=0;i<N;i++) a[i] = i;

  NS=0; NE=N  // start & end

  do{
    work(knt,NS,NE, N,a);
    knt=knt+1;
  }while (knt < 100);

}
```

Code Example 6. Code illustrates the use of the signal specifier and the stand-alone offload_wait directive.

It will also be necessary to instrument the offload directive for asynchronous execution, and wait for the offload to complete at the end of each loop.

Code example 6b shows the modifications made for offloading. The work routine is now called twice within the loop, splitting the array work among the MIC offload and the CPU. Since the work function will be executed on the MIC, the work function/subroutine must be declared with a MIC target attribute. Also, in the main program the C/C++ prototype and the Fortran external use the same attribute specification. These attributes are required whether the function/subroutine resides in the same module (file) as the main program, or not. If there are a large number of functions used in the offload region, there are compiler, loader and library options to automatically decorate the functions.

The `signal(sig1)` specifier of the offload directive instructs the compiler to do the offload asynchronously (start the offload and continue processing after the end of the directive region). The `sig1` variable is used as a handle for holding the state of the offload event, and is used at the end of the loop in an `offload_wait` directive to assure that there is no race condition between updating knt and using it. The stand-alone `offload_wait` directive before the `knt` counter increment line forces the CPU execution to wait until the offload finishes (using `sig1` as the identifier of the event to wait for).

Fortran code:

```
!Fortran

!dir$ attributes offload:mic  ::  work
subroutine work(knt, ns,ne, a)
integer :: a(*)
   do i= ns,ne
      a(i) = a(i) + 1
   end do
end subroutine

program main
!dir$ attributes offload : mic :: work
   integer,parameter :: N=100
   integer :: i, knt=1, a(N),NS,NE, sig1=1

   do i = 1,N; a(i) = i; end do

   do while (knt .lt. 10)
      NS=1; NE=N/2
      !dir$ offload      target(mic:0) signal(sig1) inout(a(1:N/2))
      call work(knt,NS,NE, a)

      NS=N/2+1; NE=N
      call work(knt,NS,NE, a)

      !dir$ offload_wait target(mic:0)   wait(sig1)
      knt=knt+1;
   end do

   do i = 1,N
      print*, i, a(i)
   end do

end program
```

Fortran Code Example 6b. Asynchronous offload of the work function/subroutine.

C code:

```
//C

__attribute__ ((target(mic:0)))  void work(int knt, int NSm, int NEm, int *a){
   int i;
   for(i=NSm;i<NEm;i++){
      a[i]= a[i] + 1;
   }
}

#define N 100
__attribute__ ((target(mic:0)))  void work(int, int, int, int *);

int main(){
int sig1, i, knt=1, *a, NSm, NEm, NSc, NEc;

  a=(int*)malloc(N*sizeof(int));
  for(i=0;i<N;i++) a[i] = i;

  do{
    NSm=0; NEm=N/2;
    #pragma offload target(mic:0) signal(&sig1)  inout(a:length(NEm))
     work(knt,NSm,NEm, a);

    NSc=N/2; NEc=N;
    work(knt,NSc,NEc, a);

    #pragma offload_wait target(mic:0) wait(&sig1)
    knt=knt+1;

  }while (knt < 10);

  for(i=0;i<N;i++){
     printf("%d  %d\n",i, a[i]);
  }

}  // CPU & MIC work on different parts of a
```

C Code Example 6b. Asynchronous offload of the work function/subroutine.

The `inout` clause of the offload statement restricts the manipulation of the `a` array on the MIC to the lower half of the array while the host manipulates the upper half. Without this restriction, there would be a race condition between the host updating elements in the upper half, and un-updated values being returned from the MIC (for the same upper half elements).

Offload Compiler and Runtime Options

By default, the Intel compilers will recognize any offload directive—no special options are required. In the following example the myprog code has offload directives:

```
ifort/icc/icpc  -O2 –xhost   myprog.f90/c/cpp
```

The –O2 and –xhost options apply to the CPU code. Default offload options include the Phi architecture (see –x in compiler guide) at the –O2 level.

The compiler can be instructed to ignore offload directives with the –no-offload option:

```
ifort/icc/icpc  -O2 –xhost  -no-offload myprog.f90/c/cpp
```

During development it is useful to observe the names and sizes of variables tagged for transfer by including the -opt-report-phase=offload option as shown here:

```
ifort/icc/icpc  -O2 –xhost  -opt-report-phase=offload myprog.f90/c/cpp
```

Specific compiler and loader options for offloaded code can be set as a comma-separated list in a string within the following compiler command options, `-offload-option,mic,compiler,"<string>"` and `–offload-option,mic,ld,"<string>"`, respectively. In the following example the host code is compiled with "-O3", and the offloaded code is compiled with "-O2 –fma" and linked with "-g".

```
ifort/icc/icpc –O3 –offload-option,mic,compiler,"-O2 –fma" \
                   –offload-option,mic,ld,"-g" prog.f90/c/cpp
```

You can use the `-watch=mic-cmd` option to report on the command line for the MIC offloads. To follow the progress of offloads, set the H_TRACE variable to a level of verbosity (0-2):

```
export H_TRACE=2
./a.out
```

Use the define macro _ _MIC_ _ (defined only for code compiled for MIC execution) for MIC-specific code, or to help determine where the code is executing. The macro is defined in C/C++ and Fortran compilations. The Fortran code snippet below will indicate where the sum was determined:

```
#ifdef _ _MIC_ _
print*, "Hello MIC reduction",sum
#else
print*, "Hello CPU reduction",sum
#endif
```

Terms

Clause: A specifier in an offload directive (sometimes use in Intel documentation).

Lexical scope: A textual block of code (body) seen by the compiler—it does not include the content of any function or subroutine (dynamic scope) called within the block.

Modifiers: Data specifiers have additional options called "modifiers" (`length`, `alloc_if` and `free_if`).

Offload region: Code block following an offload directive.

Specifier: Any of the "options" of an offload directive. These include: `target`, `in/out/inout/nocopy`, `wait`, `signal` (discussed above), and "`if`" and `mandatory`. While the Intel document calls the `in/out/inout/nocopy` "offload-parameters", we define them as "data specifiers". Data specifiers have additional options called "modifiers" (length, alloc_if and free_if).

References

http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide

http://software.intel.com/en-us/articles/webinar-get-ready-for-intel-math-kernel-library-on-intel-xeon-phi-coprocessors

http://software.intel.com/sites/default/files/MKL_for_MIC_public_webinar.pdf

Code Examples (on Stampede):
/opt/apps/intel/13/ composer_xe_2013.2.146/Samples/en_US/C++ and .../Fortran

Updates:

1/7/13 MKL_MIC_ENABLE should be set to 1 (not 1 or yes). Fixed typos.
3/8/13 Typos and word smithing.
9/28/13 Now references idev instead of srun. Mandatory clause explained. Updated asyn description of tags.
Include `inout(a:length(NEm))and inout(a(1:N/2)) in example 6b.`