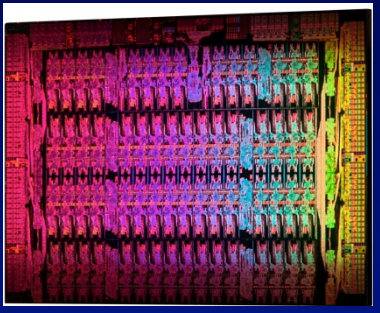


Offload Computing on Stampede

Kent Milfeld
milfeld@tacc.utexas.edu

POWERED BY XSEDE
Extreme Science and Engineering Discovery Environment

THE UNIVERSITY OF TEXAS AT AUSTIN

July, 22 2013

MIC Information

- mic-developer (programming & training tabs): http://software.intel.com/mic-developer
- Intel Programming & Compiler for MIC
- Intel Compiler Manuals: C/C++ Fortran (Key Features → Intel ® MIC Architecture)
- example code: /opt/apps/intel/13/composer_xe_2013.3.163/Samples
- Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, James Reinders. Intel Xeon Phi Coprocessor High Performance Computing, Jim Jeffers & James Reinders
- Stampede User Guide: http://www.tacc.utexas.edu/ (User Services→UserGuides→Stampede)

Offloading

- Offloading: Basic Concepts
  - Basics
  - Directive Syntax
  - Automatic Offloading (AO)
  - Compiler Assisted Offloading (CAO)
    - Directives (Code Blocks – Targets)
    - Preparation and Offload Process Steps (mechanism)
    - Data Transfers
    - Declaration for Functions and Globals, Pointer Data
    - Persistent Data
    - Asynchronous Offloading
- Offloading inside an OMP parallel region.

- Offloading: Basic Concepts
  - Basics
  - Directive Syntax
  - Automatic Offloading (AO)
  - Compiler Assisted Offloading (CAO)
    - Directives (Code Blocks – Targets)
    - Preparation and Offload Process Steps (mechanism)
    - Data Transfers
    - Declaration for Functions and Globals, Pointer Data
    - Persistent Data
    - Asynchronous Offloading
- Offloading inside an OMP parallel region.

Definition of a Node

A “node” contains a host and a MIC component

- host – refers to the Sandy Bridge component
- MIC – refers to Intel Xeon Phi co-processor cards

NODE on Stampede

host
2x Intel 2.7 GHz E5-2680
16 cores
32 GB Memory

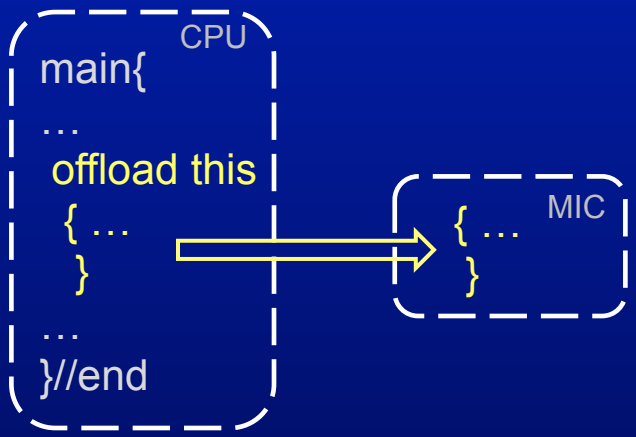
MIC
1 or 2 Intel Xeon PHI SE10P
61 cores/244 HW threads
8GB Memory

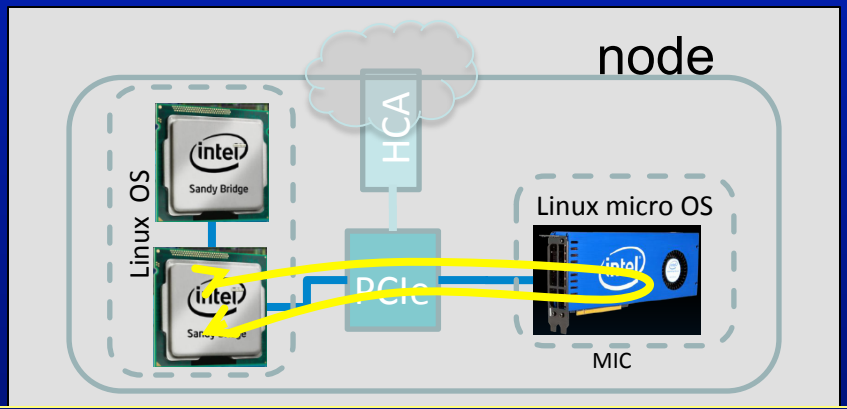
Offloading Strategy

- Think threads
  - (Whether working on a MIC, GPU, ARM, etc.)
- Options:
  - Have the MIC do all of the work
    - May be viable for low-performance-CPU – MIC solution
  - Share the work -- host and MIC
    - More reasonable for HPC system with MICs
- Great time to venture into many-core architectures
  1.) Try offloading compute-intensive section
  If it isn't threaded, make it threaded
  2.) Optimize data transfers
  3.) Split calculation & use asynchronous mechanisms

Basics: What is Offloading

- Send block of code to be executed on coprocessor (MIC).
  - Must have a binary of the code (code block or function).
  - Compiler makes the binary and stores it in the executable (a.out).
- During execution on the CPU, the "runtime" is contacted to begin executing the MIC binary at an offload point.
  - When the coprocessor is finished, the CPU resumes executing the CPU part of the code.





CPU execution is directed to run a MIC binary section of code on the MIC.

Models

- Non-Shared memory
  - Host and MIC have separate memory sub systems– think distributed memory and bit-wise data copy between platforms.
- Virtual-Shared Memory
  - C/C++; complex data structures (pointer based structures, classes, etc.) can be shared; coherency overhead.

Best: When compute complexity is $O(N^{i+1})$ and data complexity is $O(N^{i})$

Code is non-IO intensive

Offload can be done asynchronously

Basics: Directives

- Directives can be inserted before code blocks and functions to run the code on the Xeon Phi Coprocessor (the "MIC").
  - No recoding required. (Optimization may require some changes.)
  - Directives are simple, but more "details" (specifiers) can be used for optimal performance.
  - Data must be moved to the MIC
    - For large amounts of data:
      Amortize with large amounts of work.
      Keep data resident ("persistent").
      Move data asynchronously.

Basics: Simple Example

```
int main(){
  float a[10];  int i;
  #pragma offload target(mic)
  {for(i=0,i<10;i++)
     a[i]=(float) i;
  }
  #pragma offload target(mic)
   foo(a);
   printf(" %f \n",a[10]);
}
```

Insert Directives to Offload, code block or function.

```
program main
  real :: a(10)
  !dir$ offload begin target(mic)
  do i=1,10
     a(i)=I; end do
  !dir$ end offload
  !dir$ offload target(mic)
  call foo(a)
  print*, a(10)
end program
```

- Insert Offload Directive: `#pragma offload target(mic)` C/C++ / `!dir$ offload target(mic)` F
- Compile with Intel Compiler: `icc prog.c` `ifort prog.f90`
- How to turn off offloading: use `–no-offload` option

Basics: Simple OMP Example

```
#define N 10000
int main(){
  float a[N];  int i;

  #pragma offload target(mic)
  #pragma omp parallel for
  for(i=0,i<N;i++)
     a[i]=(float) i;

  printf(" %f \n",a[10]);
}
```

OMP Parallel Regions Can Be Offloaded Directly

```
program main
  integer, parameter :: N=10000
  real :: a(N)

 !dir$ offload target(mic)
 !$omp parallel do
  do i=1,N
     a(i)=I; end do

  print*, a(10)
end program
```

- OpenMP regions can be offloaded directly.
- OpenMP parallel regions can exist in offloaded code blocks or functions.

OMP: Compile & Run

- Compile on login node (as shown), or on compute node interactively (see idev in lab exercise).

```
login2$    icc –openmp –xhost –O3 omp_prog.c
login2$  ifort –openmp –xhost –O3 omp_prog.f90

login2$  idev                                    *
```

- Run on compute node (or in batch script).

```
c559-001$ export MIC_PREFIX=MIC
c559-001$ export       OMP_NUM_THREADS=16
c559-001$ export MIC_OMP_NUM_THREADS=240
c559-001$ ./a.out
```

- Use KMP_AFFINITY when thread count < 4*core count.

Tells runtime to find MIC_ prefixed variables, strip of MIC_ and use them on MIC.

"C559-001$" is the shell prompt for a compute node (host+mic) after executing *idev*

- Offloading: Basic Concepts
  - Basics
  - Directive Syntax
  - Automatic Offloading (AO)
  - Compiler Assisted Offloading (CAO)
    - Directives (Code Blocks – Targets)
    - Preparation and Offload Process Steps (mechanism)
    - Data Transfers
    - Declaration for Functions and Globals, Pointer Data
    - Persistent Data
    - Asynchronous Offloading
- Offloading inside an OMP parallel region.

Offload Directive

#pragma offload *specifier* [[,] *specifier*]
!dir$ offload "

C/C++
Fortran

specifier:

target(targ-name [*:dev#*])

if(*if-specifier*) or **mandatory**

signal(*tag*) **wait**(*tag*)

data_specifier(…)

Intel calls this an "offload-parameter".
For this training module I named it something more reasonable.

Often called "clauses".

Offload Directive

***data_specifier*:**

For explicit data transfers.

in(identifier [[,]*identifier…*] [: modifier [[,] *modifier…*])

out(“)

inout(“)

nocopy(“)

variables
arrays
…

length()
alloc_if()
free_if()
align

storage handlers

Offload Directive

Offload Directives

C/C++ starts with: #pragma _____

Fortran starts with: !dir$ _____

offload*

offload_attribute — Specifies MIC vars & functions

offload_transfer — data Host ←→MIC

offload_wait — Wait for async. offload

Stand Alone directives (no offload code)

* Fortran uses offload begin … end offload, C/C++ uses {…}

__attribute__ and __declspec "decorations" can be used in lieu of offload_attribute in C/C++. Use !dir$ attributes *list* in Fortran.

- Offloading: Basic Concepts
  - Basics
  - Directives
  - Automatic Offloading (AO)
  - Compiler Assisted Offloading (CAO)
    - Directives (Code Blocks – Targets)
    - Preparation and Offload Process Steps (mechanism)
    - Data Transfers
    - Declaration for Functions and Globals, Pointer Data
    - Persistent Data
    - Asynchronous Offloading
- Offloading inside an OMP parallel region.

Automatic Offload

- Offloads some MKL routines automatically
  - No coding change
  - No recompiling
- Makes sense with BLAS-3 type routines
  - Minimal Data $O(n^2)$, Maximal Compute $O(n^3)$
- Supported Routines (more to come)

Type	Routine
Level-3 BLAS	xGEMM, xTRSM, STRMM
LAPACK 3 amigos	LU, QR, Cholesky
Eigen Solver	

Automatic Offload

- Compile as usual, use new –mkl
  - Works with serial, OpenMP and MPI codes.
- Enable with MKL_MIC_ENABLE variable

```
login1$      ifort  -mkl -xhost -O2   app_has_MKLdgemm.f90
login1$      icc    -mkl -xhost -O2   app_has_MKLdgemm.c
...
c559-001$    export OMP_NUM_THREADS=16
c559-001$    export MKL_MIC_ENABLE=1
c599-001$    ./a.out
```

See **MKL_MIC_WORKDIVISION environment variable to set (force) a relative work load.**

- Offloading: Basic Concepts
  - Basics
  - Directives
  - Automatic Offloading (AO)
  - Compiler Assisted Offloading (CAO)
    - Directives (Code Blocks – Targets)
    - Preparation and Offload Process Steps (mechanism)
    - Data Transfers
    - Declaration for Functions and Globals, Pointer Data
    - Persistent Data
    - Asynchronous Offloading
- Offloading inside an OMP parallel region.

Compiler Assisted Offload

- Compiler looks for offload directive everywhere:
  - Before blocks, functions (subroutines), statements
  - For global variables and function declarations
  - As stand-alone directives for data transfer and waits

- Target(mic : dev_id)

target(mic) : Execute on runtime selected MIC, on cpu if error or not available
target(mic:-1) : Execute on runtime selected MIC, fail otherwise
target(mic:0-n) : Execute on device id=mod(#, no. of coprocs), fail otherwise

With more than 1 MIC use dev-id with:offload, offload_transfer, offload_wait

Compiler Assisted Offload

```
int main(){
...
  #pragma offload target(mic:0)
  {
     #pragma omp parallel for
     for (i=0; i<N;i+){
        a[i]=sin(b[i])+cos(c[i]);
     }
   }
...
}
```

```
program main
...
!dir$ offload begin target(mic:0)

!$omp parallel do
do i = 1,N
   a(i)=sin(b(i))+cos(c(i))
end do
!dir$ end offload
...
 end program
```

- Data (a,b, and c) within lexical scope are moved implicitly.

- C/C++ use {…} (curly braces) to mark a block
- Fortran use begin and !dir$ end offload to mark block

The Offload Preparation

- Code is instrumented with directives.
- Compiler creates a CPU binary and a MIC binary for offloaded code block.
- Loader places both binaries in a single file. (→ a.out)
- During CPU execution of the application an encountered offload code block is executed on a coprocessor (through runtime), subject to the constraints of the target specifier…

The Offload Mechanism

- The basic operations of an offload rely on interaction with the runtime to:

Detect a target phi coprocessor

Allocate memory space on the coprocessor

Transfer data from the host to the coprocessor

Execute offload binary on coprocessor

Transfer data from the coprocessor back to the host

Deallocate space on coprocessor

Binaries are moved on first offload.

Data Transfers

- If you know the intent of data usage, minimize unnecessary transfers with in/out/inout data specifiers.

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

#pragma offload target(mic)     in( b,c )   // Only copy b and c into MIC
#pragma offload target(mic)    out(a    )   // Only return a
#pragma offload target(mic)  inout(d    )   // Default, copy into and out of
```

Data Transfers

```
int main(){
...
  #pragma offload target(mic) \
                      in(b,c) out(a)
  {
     #pragma omp parallel for
     for (i=0; i<N;i+){
        a[i]=sin(b[i])+cos(c[i]);
     }
   }
...
}
```

```
program main
...
!dir$ offload begin target(mic) &
                      in(b,c) out(a)

!$omp parallel do
do i = 1,N
  a(i)=sin(b(i))+cos(c(i))
end do
!dir$ end offload
...
 end program
```

Offload Functions, Globals & Pointer Data

- "Decorate" all functions used in offloads with a target "attribute".
- Likewise with globals

_ _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>	F90

Offload Functions, Globals & Pointer Data

C/C++

```
__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);
}
```

F90

```
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
```

Offload Functions, Globals & Pointer Data

- Offload attributes can be applied to an entire file through a compiler option:

C/C++

```
icpc | icc | f90   -c -offload-attribute-target=mic   my_fun.cpp | c | f90
icpc | icc | f90       my_fun.o                      my_app.cpp | c | f90
```

- C/C++ has file scoping, FORTRAN does not:

C/C++

```
#pragma offload_attribute(push, target(mic))
    void fun1(int i) {i=i+1;}
    void fun2(int j) {j=j+2;}
#pragma offload_attribute(pop)
```

FORTRAN

```
module my_globs
  !dir$ options /offload_attribute_target=mic
  real, allocatable :: in1(:), in2(:), out1(:), out2(:)
  !dir$ end options
end module
```

Offload Functions, Globals & Pointer Data

- C pointer to contiguous data requires length modifier– (default copy is 1 element).
- Not required for Fortran allocated arrays.

```
...
a=(double *) malloc(N   *sizeof(double));
b=(double *) malloc(N   *sizeof(double));
c=(double *) malloc(N   *sizeof(double));
d=(double *) malloc(M   *sizeof(double));
e=(double *) malloc(N*2*sizeof(double));
...
#pragma offload target(mic:0)    in( a,b,c : length(    N) )   // pointers a, b & c, length N
#pragma offload target(mic:0)  out(      d : length(    M) )   // pointer d has      length M
#pragma offload target(mic)   inout(     e : length(2*N) )   // pointer e has length of  Nx2
```

Alignment might be important

Persistent Data

- Default implicit and explicit behavior:
  allocate space for all data before offload, and
  deallocate (free) on offload completion.

alloc_if(*logic_expression* **)** – if true allocate space at begin
free_if(*logic_expression* **)** – if true free space at end

- The offload_transfer directive allows data management (data specifiers) without a code block. It is a stand-alone directive.

Persistent Data

- Fortran and C/C++ syntaxes are identical, except:
  - Sentinels are different: #pragma versus!dir$
  - Truth variables: Fortran: logical .true./.false. C/C++ int 1/0

#pragma offload data_specifier(identifier(s): alloc_if(TorF) free_if(TorF))

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

… == target(mic)

Alloc/Free Truth Table

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)

Asynchronous Offloading

- Default behavior: CPU process waits for offload to complete.
- Signal and wait specifiers allow CPU to continue executing after the offload code block, once the runtime is notified to perform the offload (i.e. offload becomes asynchronous).
- Offload_wait is a stand-alone directive (no code block).

Syntax:

```
#pragma offload target(mic[:#id]) … signal(tag_list)
#pragma offload target(mic[:#id]) …   wait(tag_list)
#pragma offload_wait              …   wait(tag_list)

!dir$  offload target(mic[:#id])  …  signal(tag_list)
!dir$  offload target(mic[:#id])  …   wait(tag_list)
!dir$  offload_wait               …   wait(tag_list)
```

where tag_list is a set of comma separated variables

Asynchronous Offloading

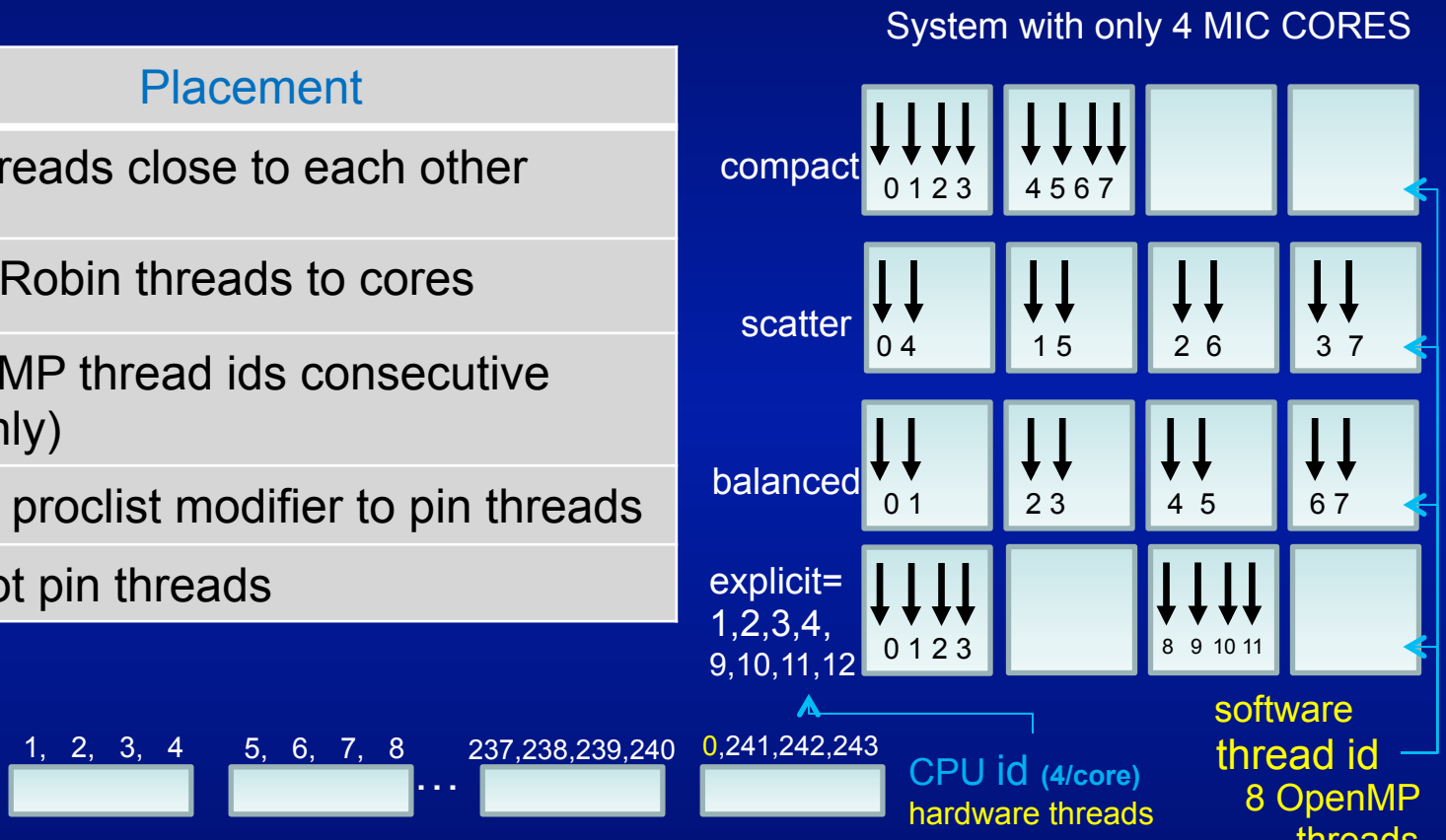
```
 #define N 10000
__attribute__ ((target(mic:0)))
void work(int knt, int M,int N, int *a);
int main(){
int sig1, i, knt=1, a[N], NS, NE;

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

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

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

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

 }while (knt < 10);

}
```

- Offload events are identified by a tag (variable address).
  - F90: signal(var)
  - C/C++: signal(&var)
- Wait/signal can have multiple tags.
- Directives can have wait and signal specifiers.

Offload Thread Placement

Controlled through environment variable: KMP_AFFINITY=<type>

Type	Placement
compact	pack threads close to each other
scatter	Round-Robin threads to cores
balanced	keep OMP thread ids consecutive (MIC only)
explicit	use the proclist modifier to pin threads
none	does not pin threads



Offload automatically avoids core 60 (HW threads 0,241,242,243), and with scatter/compact.

Be careful if you pin threads with *explicit*, offload communication/transfers occur on core 60.

- Offloading: Basic Concepts
  - Basics
  - Directives
  - Automatic Offloading (AO)
  - Compiler Assisted Offloading (CAO)
    - Directives (Code Blocks – Targets)
    - Preparation and Offload Process Steps (mechanism)
    - Data Transfers
    - Declaration for Functions and Globals, Pointer Data
    - Persistent Data
    - Asynchronous Offloading
  - Offloading inside an OMP parallel region.

Offload in parallel region

```
omp_set_nested(1);
omp_set_max_active_levels(2);
omp_set_num_threads(2);
  #pragma omp parallel
  {  printf("reporting in from %d\n", \
                omp_get_thread_num());

    #pragma omp sections
    {
      #pragma omp section
      {
        #pragma offload target(mic)
        foo(i);
      }
      #pragma omp section
      {
        #pragma omp parallel for num_threads(3)
        for(i=0;i<3;i++) {bar(i);}
      }
    }
  }
```

Sections allows 1 generating thread in each section.

Nested level re-defines a thread team with new thread ids. (Worksharing team is no longer dependent upon original parallel region team size.) Scheduling can be static!

.

Compiler Options and Env Vars

Compiler	Purpose
-no-offload	Ignore offload directives
-offload-attribute-target=mic	Flag every global data object and routine with the offload attribute
-opt-report-phase=offload	Optimization phase report for offload
-offload-option,mic,compiler,"option list"	Compiler options for MIC
-offload-option, ld,compiler,"option list"	Loader options for MIC

Environment Variable	Purpose
MIC_ENV_PREFIX	(usually =MIC) Controls variables passed to MIC.
OFFLOAD_REPORT	(=1|2|3) Controls printing offload execution time (sec), and the amount of data transferred (bytes).
MIC_STACKSIZE	Specifies the stack size of the main thread for the offload. (default =12M)
MKL_MIC_ENABLE	(=1) Sets automatic offloading on.
MKL_MIC_WORKDIVISION MKL_HOST_WORKDIVISION	Sets fraction of automatic offload work for MIC/HOST.

References

In these Compiler User Guides for offload details GO TO:
Key Features→Intel MIC Architecture→Programming for Intel MIC Architecture

- http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/fortran-lin/index.htm
- http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm

Intel MIC Programming and Computing

- http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture

Developer's Guide

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

MKL

- 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