Offloading

Kent Milfeld
milfeld@tacc.utexas.edu

Stampede Training, January 11, 2013
MIC Information

- **Stampede User Guide**: http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide

- **TACC Advanced Offloading**: Search and click on “Advanced Offloading document” in the Stampede User Guide.

- **Intel Programming & Compiler for MIC**

- **Intel Compiler Manuals**: C/C++  Fortran

- **Example code**: /opt/apps/intel/13/composer_xe_2013.1.117/Samples/en_US/
• 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 Global, Pointer Data
    • Persistent Data
    • Asynchronous Offloading

• Offloading inside an OMP parallel region.
• 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 Global, Pointer Data
    • Persistent Data
    • Asynchronous Offloading
• Offloading inside an OMP parallel region.
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.
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.
Offloading
Basics
Example

Basics: Simple Example

Insert Offload Directive:

```
it 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]);
}
```

Compile with Intel Compiler:

```
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
```

How to turn off offloading:

```
#pragma offload target(mic) C/C++
!dir$ offload target(mic) F
```

use –no-offload option
• 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 srun in lab exercise).

- Run on compute node or in batch script:

  ```
  login2$  icc  -openmp  -xhost  -O3  omp_prog.c
  login2$  ifort  -openmp  -xhost  -O3  omp_prog.f90
  c559-001$  export  OMP_NUM_THREADS=16
  c559-001$  export  MIC_OMP_NUM_THREADS=240
  c559-001$  export  MIC_PREFIX=MIC
  c559-001$  ./a.out
  ```

- Use KMP_AFFINITY when thread count < core count.

- Tells runtime on MIC to look for (use) MIC prefixed variables instead.

“C559-001$” is the shell prompt for an interactive srun.
• 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 Global, Pointer Data
    • Persistent Data
    • Asynchronous Offloading

• Offloading inside an OMP parallel region.
**#pragma offload specifier [ [ , ] specifier ]**  

C/C++

**!dir offload**  

Fortran

**specifier:**

- **target**( targ-name [:dev#] )
- **if**( if-specifier )
- **signal**( tag )  
  **wait**( tag )
- **data_specifier**( ...)

Often called “clauses”.

Intel calls this a “offload-parameter”.  
For this training module I named it something more reasonable.
Offload Directives

**data_specifier**:  

- **in**
- **out**
- **inout**
- **nocopy**

Variables, arrays...

- **length()**
- **alloc_if()**
- **free_if()**

Storage handlers
• 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 Global, 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)

<table>
<thead>
<tr>
<th>Type</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-3 BLAS</td>
<td>xGEMM, xTRSM, STRMM</td>
</tr>
<tr>
<td>LAPACK 3 amigos</td>
<td>LU, QR, Cholesky</td>
</tr>
</tbody>
</table>
Automatic Offload

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

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

```bash
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
```
• 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 Global, Pointer Data
    • Persistent Data
    • Asynchronous Offloading
• Offloading inside an OMP parallel region.
Compiler Assisted Offload

- A **code block** is assigned to execute on a coprocessor by a **directive**:

  ```
  #pragma offload target(mic[:dev_id]) … C/C++
  !dir offload target(mic[:dev_id]) … FORTRAN
  ```

**code block:**

- single or multiple program statements
  ```
  C/C++: {...} F90: $dir offload begin...$dir end offload
  ```
- can be a function or subroutine call
- can be an OpenMP construct

**target(mic[:dev_id]):**

- **target(mic):** offload on MIC if available, use CPU if MIC fails
- **target(mic[:dev_id]):** only offload, on MIC identified as dev_id (0, 1, …)
int main()
{
    ...
    #pragma offload target(mic)
    {
        #pragma omp parallel for
        for (i=0; i<N;i+)
        {
            a[i]=sin(b[i])+cos(c[i]);
        }
    }
    ...
}

program main
{
    ...
    !$omp parallel do
    do i = 1,N
        a(i)=sin(b(i))+cos(c(i))
    end do
    !$omp end parallel do
    ...
    end program
}

• 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 coprocessor binary (and a CPU binary) for offloaded code block.
- Loader places both binaries in a single file.
- During CPU execution of the application an encountered offload code block is executed on a coprocessor, 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
Data Transfers

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

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

#pragma offload target(mic:0) in(b,c) // Only copy b and c into MIC
#pragma offload target(mic:0) out(a) // Only return a
#pragma offload target(mic) inout(d) // Default, copy into and out of
```
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
}
“Decorate” all functions used in offloads with a target “attribute”.

Likewise with globals

```c
_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++**

```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**

```fortran
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

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

```c
... 
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));
...
#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

- Default behavior is to allocate space for all data before offload, and to deallocate (free) on offload completion.
- Allocation & Deallocation can be controlled with alloc_if and free_if modifiers.
- The offload_transfer directive allow data management (data specifiers) without a code block. It is a stand-alone directive.
• 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

```c
#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
```
## Alloc/Free Truth Table

<table>
<thead>
<tr>
<th>Allocation Operation</th>
<th>Deallocation (Free) Operation</th>
<th>Operations Performed (Use Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alloc_if(true)</code></td>
<td><code>free_if(true)</code></td>
<td>This is the default when no storage operations are specified. Allocate space at beginning, free at end.</td>
</tr>
<tr>
<td><code>alloc_if(true)</code></td>
<td><code>free_if(false)</code></td>
<td>Allocate space, don’t free (make space available on device, and retain for future use).</td>
</tr>
<tr>
<td><code>alloc_if(false)</code></td>
<td><code>free_if(true)</code></td>
<td>don’t allocate, but free (reuse device storage, but will not need later)</td>
</tr>
<tr>
<td><code>alloc_if(false)</code></td>
<td><code>free_if(false)</code></td>
<td>don’t allocate, don’t free (reuse device storage, and leave for future use)</td>
</tr>
</tbody>
</table>
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 (=async offload).
- **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

Offload events are identified by a tag (variable address).
F90: \texttt{signal( \ var)}
C/C++: \texttt{signal(\&var)}

Wait/signal can have multiple tags.

Directives can have wait and signal specifiers.

```c
#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{
        NS=0; NE=N/2;
        #pragma offload target(mic:0) signal(&sig1)
        work(knt,NS,NE, N,a);

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

        #pragma offload_wait target(mic:0) wait(&sig1)
        knt=knt+1;
    }while (knt < 10);
}
```
• **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 Global, Pointer Data
    • Persistent Data
    • Asynchronous Offloading
  – Offloading inside an OMP parallel region.
Heterogeneous Computing, Sequential

Offloading
- Offload in OMP
- Inside Parallel Region

MPI process, master thread

Generate parallel region

idle threads

workshare on cpu

#pragma omp parallel
{
    #pragma omp single
    #pragma offload target(mic)
    { foo(); }

    #pragma omp for
    for(i=0; i<N; i++)
    {...}
}

C/C++

F90

idle threads

!$omp parallel
!$omp single
!DIR$ offload target(mic)
call foo();
!$omp end single

!$omp do
    do i=1,N; ...
    end do
!$omp end parallel

F90
Heterogeneous Computing, Concurrent

Offloading

- Offload in OMP
- Inside Parallel Region

MPI process, master thread

Generate parallel region

offload single nowait

workshare on cpu

assist when done in single

### C/C++

```c
#pragma omp parallel
{
    #pragma omp single nowait
    #pragma offload target(mic)
    { foo(); }

    #pragma omp for schedule(dynamic)
    for(i=0; i<N; i++)
    {...}
}
```

### F90

```fortran
!$omp parallel
!$omp single
!DIR$ offload target(mic)
    call foo();
!$omp end single nowait

!$omp do schedule(dynamic)
    do i=1,N; ...
    end do
!$omp end parallel
```
#include <omp.h>
#include <stdio.h>

int main() {
    const int N=100000000;
    int i, nt, N_mic, N_cpu;
    float *a;

    a = (float *) malloc(N*sizeof(float));
    for(i=0; i<N; i++) a[i]=-1.0; a[0]=1.0;

    N_mic = N/2; N_cpu = N/2;
    nt = 16; omp_set_num_threads(nt);

    #pragma omp parallel
    {
        #pragma omp single nowait
        {
            #pragma offload target(MIC:0) out(a:length(N_MIC))
            #pragma omp parallel for
            for(i=0; i<N_mic; i++) { a[i]=(float)i; } 
        } 

        #pragma omp for schedule(dynamic,N/nt)
        for(i=N_cpu; i<N; i++) { a[i]=(float)i; } 
    }

    printf("a[0],a[N-1] %f %f\n",a[0],a[N-1]);
}
• OpenMP 3.0 supports nested parallelism, older implementations may ignore the nesting and serialize inner parallel regions.

• A nested parallel region can specify any number of threads to be used for the thread team, new id’s are assigned. Scheduling: static, etc.
omp_set_nested(1);
omp_set_max_active_levels(2);
omp_set_num_threads(2);

#pragma omp parallel private(id)
{  printf("reporting in from %d\n", \  omp_get_thread_num());

#pragma omp sections nowait
{
    #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!
References

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


Intel MIC Programming and Computing


Developer’s Guide


MKL