Intel Xeon Phi MIC
Offload Programming Models

Doug James
Oct 2013
Key References

• Jeffers and Reinders, *Intel Xeon Phi...*
  – but some material is no longer current

• Intel Developer Zone

• Stampede User Guide and related TACC resources
  – Search User Guide for "Advanced Offload" and follow link

Other specific recommendations throughout this presentation
Overview

Basic Concepts
Three Offload Models
Issues and Recommendations

Source code available on Stampede:

```
tar xvf ~train00/offload_demos.tar
```

Project codes: **TG-TRA120007** (XSEDE Portal), **20131004MIC** (TACC Portal)
A program running on the host “offloads” work by directing the MIC to execute a specified block of code. The host also directs the exchange of data between host and MIC.

Ideally, the host stays active while the MIC coprocessor does its assigned work.
Offload Models

• Compiler Assisted Offload
  – Explicit
    • Programmer explicitly directs data movement and code execution
  – Implicit
    • Programmer marks some data as “shared” in the virtual sense
    • Runtime automatically synchronizes values between host and MIC

• Automatic Offload (AO)
  – Computationally intensive calls to Intel Math Kernel Library (MKL)
  – MKL automatically manages details
  – More than offload: work division across host and MIC!
Explicit Model: Direct Control of Data Movement

- aka Copyin/Copyout, Non-Shared, COI*
- Available for C/C++ and Fortran
- Supports simple (“bitwise copyable”) data structures (think 1d arrays of scalars)

*Coprocessor Offload Infrastructure
program main

  use omp_lib

  integer :: nprocs

  nprocs = omp_get_num_procs()
  print*, "procs: ", nprocs
end program

#include <stdio.h>
#include <omp.h>

int main( void ) {

  int totalProcs;

  totalProcs = omp_get_num_procs();

  printf( "procs: %d\n", totalProcs );
  return 0;
}

Simple Fortran and C codes that each return "procs: 16" on Sandy Bridge host...
Add a one-line directive/pragmas that offloads to the MIC the one line of executable code that occurs below it...

...codes now return "procs: 240"...
Don't even need to change the compile line...
Explicit Offload

Not asynchronous (yet): the host pauses until MIC is finished.
Explicit Offload

Can offload a block of code (generally safer than the one-line approach)...

F90

!dir$ offload begin target(mic)
   nprocs = omp_get_num_procs()
   maxthreads = omp_get_max_threads()
!dir$ end offload

C/C++

#pragma offload target(mic)
{
   totalProcs =omp_get_num_procs();
   maxThreads =omp_get_max_threads();
}

off02block
...or an OpenMP region defined by an omp directive...

```f90
program main

    integer, parameter :: N = 500000  ! constant
    real            :: a(N)        ! on stack

!dir$ offload target(mic)
 !$omp parallel do
    do i=1,N
      a(i) = real(i)
    end do
 !$omp end parallel do
...
```

```c/c++
int main( void ) {

    double a[500000];
    // on the stack; literal here is important
    int i;

    #pragma offload target(mic)
    #pragma omp parallel for
    for ( i=0; i<500000; i++ ) {
        a[i] = (double)i;
    }
...
```
 integer function successor( m )
  ...

program main
  ...
  integer :: successor
  ...

!dir$ offload target(mic)
  n = successor( m )

...or procedure(s) defined by the programmer
(though now there's another step)...

int successor( int m );
void increment( int* pm );

int main( void ) {
  int i;

  #pragma offload target(mic)
  {
    i = successor( 123 );
    increment( &i );
  }

C/C++


The code snippet demonstrates explicit offload in both C/C++ and Fortran. In C/C++, the `__declspec( target(mic) )` attributes are used to mark functions and variables for offload to a specific device, in this case, a MIC (Many Integrated Core) processor. The `#pragma offload target(mic)` directive is used to offload the indicated code block to the MIC device.

In Fortran, the `__declspec` attribute is used similarly to mark functions as targetable for offload, and the `#pragma offload target(mic)` directive is used to offload the code to the MIC device.

The code also shows how to define integer functions and variables, and how to call these functions within the main program, offloading the computation to the MIC.

...mark prototypes to tell compiler to build executable code on both sides...
Controlling the Offload

Additional decorations (clauses, attributes, specifiers, keywords) give the programmer a high degree of control over all steps in the process.
#pragma offload target(mic)
#pragma omp parallel for
for (i = 0; i < 500000; i++)
    a[i] = (double)i;

"target(mic)" means "find a MIC, any ol' MIC"...
#pragma offload
target(mic:0)
#pragma omp parallel for
for (i=0; i<500000; i++) {
a[i] = (double)i;
}

"target(mic:0)" or "target(mic:i)" means "find a specific MIC"...
double a[100000], b[100000], c[100000], d[100000];

// on the stack; literal is necessary for now
...

#pragma offload target(mic) \ 
  in( a ), out( c, d ), inout( b )

#pragma omp parallel for
for ( i=0; i<100000; i++ ) {
    c[i] = a[i] + b[i];
    d[i] = a[i] - b[i];
    b[i] = -b[i];
}

...
real, allocatable :: a(:), b(:)  
integer, parameter :: N = 5000000  
allocate( a(N), b(N) )  
...

! Fortran allocatable arrays don't need length attribute...
!dir$ offload target(mic)
   &
in( a : alloc_if(.true.) free_if(.true.) ), &
out( b : alloc_if(.true.) free_if(.false.) )
!$omp parallel do
   do i=1,N
      b(i) = 2.0 * a(i)
   end do
!$omp end parallel do

…manage MIC memory and its association with memory on the host...

int N = 5000000;
double *a, *b;

a = ( double* ) memalign( 64, N*sizeof(double) );
b = ( double* ) memalign( 64, N*sizeof(double) );
...
#pragma offload target(mic)
  
in( a : length(N) alloc_if(1) free_if(1) ), 
out( b : length(N) alloc_if(1) free_if(0) )
#pragma omp parallel for
   for ( i=0; i<N; i++ ) {
      b[i] = 2.0 * a[i];
   }
real, allocatable :: a(:), b(:)  
integer, parameter :: N = 5000000  
allocate( a(N), b(N) )  
...

! Fortran allocatable arrays don't need length attribute...
!dir$ offload target(mic) &  
in( a : alloc_if(.true.) free_if(.true.) ), &  
out( b : alloc_if(.true.) free_if(.false.) )  
!$omp parallel do  
do i=1,N  
  b(i) = 2.0 * a(i)  
end do  
!$omp end parallel do

...Dynamically allocated arrays in C/C++ require an additional "length" attribute...

int N = 5000000;
double *a, *b;

a = ( double* ) memalign( 64, N*sizeof(double) );
b = ( double* ) memalign( 64, N*sizeof(double) );
...
#pragma offload target(mic)  
\in( a : length(N) alloc_if(1) free_if(1) ), \
out( b : length(N) alloc_if(1) free_if(0) )  
#pragma omp parallel for  
for ( i=0; i<N; i++ ) {  
  b[i] = 2.0 * a[i];  
}
integer :: n = 123

!dir$ offload begin target(mic:0) signal( n )
    call incrementSlowly( n )
!dir$ end offload
...

print *, " n: ", n

...Asynchronous offload with “signal”: work on host continues while offload proceeds...

int n = 123;

#pragma offload target(mic:0) signal( &n )
    incrementSlowly( &n );
...

printf( "\n\tn = %d \n", n );
integer :: n = 123

!dir$ offload begin target(mic:0)  signal( n )
   call incrementSlowly( n )
!dir$ end offload

...
!dir$ offload_wait target(mic:0)  wait( n )

printf( "n = %d
", n );

...offload_wait pauses the host but initiates no new work on MIC...
int n = 123;

#pragma offload target(mic:0)  signal( &n )
  call incrementSlowly( &n )
#pragma offload target(mic:0)  wait( &n )
  printf( "\n\ntprocs: %d\n", omp_get_num_procs() )
  call flush(0)

print *, " n: ", n

...classical offload
(as opposed to
offload_wait)
will offload the next
line/block of code...

...both constructs need a
wait() clause with tag
...offload_transfer is a data-only offload (no executable code sent to MIC)...

...use it to move data and manage memory (alloc and free)...

these examples are asynchronous
Detecting/Monitoring Offload

- **export OFFLOAD_REPORT=2**  # or 1, 3
- Compile time info: **-opt-report-phase=offload**
- **__MIC__** macro defined on device
  - can be used for conditional compilation
  - use only within offloaded procedure
  - use capitalized “F90” suffix to pre-process during compilation
- **ssh mic0** (not mic:0) and run top
  - offload processes owned by “micuser”
Other Key Environment Variables

**OMP_NUM_THREADS**
- default is 1; that’s probably not what you want!

**MIC_OMP_NUM_THREADS**
- default behavior is 244 (var undefined); you definitely don’t want that

**MIC_STACKSIZE**
- default is only 12MB

**MIC_KMP_AFFINITY** and other performance-related settings
Offload: making it worthwhile

- Enough computation to justify data movement
- High degree of parallelism
  - threading, vectorization
- Work division: keep host and MIC active
  - asynchronous directives
  - offloads from OpenMP regions
- Intelligent data management and alignment
  - persistent data on MIC when possible

Automatic Offload (AO)

- Feature of Intel Math Kernel Library (MKL)
  - growing list of computationally intensive functions
  - xGEMM and variants; also LU, QR, Cholesky
  - kicks in at appropriate size thresholds
    - (e.g. SGEMM: (M,N,K) = (2048, 256, 2048)

- Essentially no programmer action required
  - more than offload: work division across host and MIC
...call one of the supported MKL functions for sufficiently large matrices...

#include "mkl.h"

...m = 8000;
p = 9000;
n = 10000;
...
cblas_dgemm(
    CblasRowMajor, CblasNoTrans, CblasNoTrans,
    m, n, p, alpha, A, p, B, n, beta, C, n );
Automatic Offload

...use Intel compiler and link to MKL...

...ldd should show libmkl_intel_thread...

```c
#include "mkl.h"

... 
m = 8000;
p = 9000;
n = 10000;
...

cblas_dgemm(
    CblasRowMajor, CblasNoTrans, CblasNoTrans,
    m, n, p, alpha, A, p, B, n, beta, C, n );
```

```fortran
CALL DGEMM( 'N','N',M,N,P,ALPHA,A,M,B,P,BETA,C,M )
```

```c
ifort -openmp -mkl main.f

... 

M = 8000 
P = 9000 
N = 10000 
...
```

```bash
...use Intel compiler and link to MKL...

...ldd should show libmkl_intel_thread...

```
Automatic Offload

- Set at least three environment variables before launching your code:

  ```
  export MKL_MIC_ENABLE=1
  export OMP_NUM_THREADS=16
  export MIC_OMP_NUM_THREADS=240
  ```

- Other environment variables provide additional fine-grained control over host-MIC work division et al.

  [Link](http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/GUID-3DC4FC7D-A1E4-423D-9C0C-06AB265FFA86.htm)
MKL Offload: Other Opportunities

- Apps that call MKL “under the hood” can exploit AO
  - Need to build with Intel and link to threaded MKL
    - In other words, use `-mkl` or `-mkl=parallel`; do not use `-mkl=sequential`
  - Matlab on Stampede:
    - `export BLAS_VERSION=$TACC_MKL_LIB/libmkl_rt.so`
  - AO for R temporarily available with "module load R_mkl"
    - New AO-enabled parallel R coming soon
  - AO for Python: coming soon to Stampede
- Can also explicitly offload MKL functions
Implicit Offload: Virtual Shared Memory

- aka Shared Memory, MYO*
- Programmer marks data as shared between host and MIC; runtime manages synchronization
- Supports “arbitrarily complex” data structures, including objects and their methods
- Available only for C/C++

*”Mine-Yours-Ours”
Implicit Offload

_Cilk_shared marks global data as usable and synchronized between host and MIC. Runtime handles the details.

```c
int _Cilk_shared mySharedInt;
COrderedPair _Cilk_shared mySharedP1;
```
Implicit Offload

_Cilk_shared also marks functions as suitable for offload. Signatures in prototypes and definitions determine how shared and unshared functions operate on shared data.

```c
int _Cilk_shared mySharedInt;
COrderedPair _Cilk_shared mySharedP1;

int _Cilk_shared incrementByReturn( int n );
void _Cilk_shared incrementByRef( _Cilk_shared int& n );
void _Cilk_shared modObjBySharedPtr(
    COrderedPair _Cilk_shared *ptrToShared );
```

C/C++ only
### Implicit Offload

_Cilk_offload executes a shared function on MIC (does not operate on a block of code)

```c
int _Cilk_shared mySharedInt;
COrderedPair _Cilk_shared mySharedP1;

int _Cilk_shared incrementByReturn( int n );
void _Cilk_shared incrementByRef( _Cilk_shared int& n );
void _Cilk_shared modObjBySharedPtr(
    COrderedPair _Cilk_shared *ptrToShared );

...
mySharedInt
    = _Cilk_offload incrementByReturn( mySharedInt );
_Cilk_offload modObjBySharedPtr( &mySharedP1 );
```
Implicit Offload

But the devil’s in the details...

```c
int _Cilk_shared mySharedInt;
COrderedPair _Cilk_shared mySharedP1;

int _Cilk_shared incrementByReturn( int n );
void _Cilk_shared incrementByRef( _Cilk_shared int& n );
void _Cilk_shared modObjBySharedPtr(
    COrderedPair _Cilk_shared *ptrToShared );
...
mySharedInt
    = _Cilk_offload incrementByReturn( mySharedInt );
_Cilk_offload modObjBySharedPtr( &mySharedP1 );
```
Implicit Offload: Issues

- Shared data must be global
- Shared vs unshared datatypes
  - need for casting and overloading (equality, copy constructors)
- Special memory managers
  - “placement new” to share STL classes
- Infrastructure less stable and mature
  - Intel sample code available, but other resources are sparse
  - we all have a lot to learn about this
- By its nature a little slower than explicit offload
Offload: Issues and Gotchas

• Fast moving target
  – Functionality/syntax varies across compiler versions
  – Documentation often lags behind ground truth

• First offload takes longer
  – Consider an untimed initMIC offload

• Memory limits
  – ~6.7GB available for heap; 12MB default stack

• File I/O essentially impossible from offload region
  – console output ok; flush buffer

• Optional offload in transition
  – -no-offload compiler flag works on Stampede
Summary

• Offload may be for you if your app is...
  – computationally intensive
  – highly parallel (threading, vectorization)

• Best practices revolve around...
  – asynchronous operations
  – intelligent data movement (persistence)

• Three models currently supported
  – explicit: simple data structures
  – automatic: computationally-intensive MKL calls
  – implicit: complex data structures (objects and their methods)
Exercise Options (pick and choose)

• Option A: `tar xvf ~train00/offload_lab.tar`
  - Exercise 1: Simple Offload Examples
  - Exercise 2: Data Transfer Optimization
  - Exercise 3: Concurrent and Asynchronous Offloads

• Option B: `tar xvf ~train00/offload_demos.tar`
  - Explicit offload: exercises based on TACC examples from presentation
  - Automatic offload: exercises based on Intel examples from presentation

Project codes: TG-TRA120007 (XSEDE Portal), 20131004MIC (TACC Portal)
Doug James
djames@tacc.utexas.edu
(512) 471-0696

For more information:
www.tacc.utexas.edu
License

© The University of Texas at Austin, 2013

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/

When attributing this work, please use the following text: