Intel Xeon Phi MIC
Offload Programming Models

Doug James
Dec 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  - but some material is no longer current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 code: 20131204MIC (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 (CAO)**
  - 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
Simple Fortran and C codes that each return "procs: 16" on Sandy Bridge host…
Add a one-line directive/pragama 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…

F90

```fortran
program main
    use omp_lib
    integer :: nprocs
    !dir$ offload target(mic)
    nprocs = omp_get_num_procs()
    print*, "procs: ", nprocs
end program
```

C/C++

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

int main( void ) {
    int totalProcs;
    #pragma offload target(mic)
    totalProcs = omp_get_num_procs();
    printf( "procs: %d\n", totalProcs );
    return 0;
}
```

Explicit Offload

- ifort -openmp off01simple.f90
- icc -openmp off01simple.c

Don't use "-mmic"
Not asynchronous (yet): the host pauses until MIC is finished.
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();
}
Explicit Offload

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

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
...

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;
    }
...

off03omp
integer function successor( m )
...

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

!dir$ offload target(mic)
n = successor( m )
...mark prototypes to tell compiler to build executable code on both sides...

```c
int main( void ) {
    int i;
    #pragma offload target(mic)
    {
        i = successor( 123 );
        increment( &i );
    }
}
```
module mymodvars
  !dir$ attributes offload:mic :: mymoduleint
  integer :: mymoduleint
end module mymodvars

program main
  use mymodvars
  implicit none

  integer :: mylocalint = 123
  integer, save :: mysaveint !no decoration required

  __declspec( target(mic) ) int myGlobalInt;

  int main( void ) {
    int myLocalInt = 123;
    __declspec( target(mic) ) static int myStaticInt;
Controlling the Offload

Additional decorations (clauses, attributes, specifiers, keywords) give the programmer a high degree of control over all steps in the process.
"target(mic)" means "find a MIC, any ol' MIC"...
"target(mic:0)" or "target(mic:i)" means "find a specific MIC"...

```c
#pragma offload target(mic:0)
#pragma omp parallel for
for ( i=0; i<500000; i++ ) {
    a[i] = (double)i;
}
```
integer, parameter :: N = 100000  ! constant
real :: a(N), b(N), c(N), d(N)  ! on stack
...
dir$ offload target(mic) &
in( a ), out( c, d ), inout( b )
!$omp parallel do
do i=1,N
  c(i) = a(i) + b(i)
  d(i) = a(i) - b(i)
  b(i) = -b(i)
end do
!$omp end parallel do

control data transfer
between host and MIC...

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

! 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 dynamically allocated memory on the host...

```c
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 : alloc_if(1) free_if(1) ),  
  out( b : 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(:)                (see source file for
integer, parameter :: N = 5000000         alignment directives)
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

...

printf( "n = %d 
", n );
```c
int n = 123;

#pragma offload target(mic:0) signal( &n )
    call incrementSlowly( &n );
#pragma offload_wait target(mic:0) wait( &n )

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

...offload_wait pauses the host but initiates no new work on MIC...
integer :: n = 123

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

...

!dir$ offload begin target(mic:0) wait( n )
   print *, " procs: ", omp_get_num_procs()
   call flush(0)
!dir$ end offload

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

int n = 123;

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

...

#pragma offload target(mic:0) wait( &n )
{
   printf( "\n\tprocs: %d\n", omp_get_num_procs() );
   fflush(0);
}

printf( "\n\tn = %d \n", n );
 Explicit Offload

...offload_transfer is a data-only offload (no executable code sent to MIC)...

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

F90

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, 2048, 256)

• 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...

```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 );
```
Automatic Offload

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

...ldd should show libmkl_intel_thread...

#include "mkl.h"

M = 8000
P = 9000
N = 10000

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

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

ifort -openmp -mkl main.f

CALL DGEMM( 'N','N',M,N,P,ALPHA,A,M,B,P,BETA,C,M )
Automatic Offload

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

```bash
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.

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 );
```
**Implicit Offload**

\[\text{C/C++ only}\]

```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 );
```

_\texttt{Cilk\_offload}\_ executes a shared function on
MIC (does not operate on a block of code)
Implicit Offload

```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 code: 20131204MIC
For more information:
www.tacc.utexas.edu

Doug James
djames@tacc.utexas.edu
(512) 471-0696
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: “Intel Xeon Phi MIC: Offload Programming Models”, Texas Advanced Computing Center, 2013. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License”