Bring on the Stampede
Coding with the Xeon Phi

Scientific Software Days

December 17, 2012
ACES

Lars Koesterke

Texas Advanced Computing Center

Stampede

What do we do with 6400 MIC cards

Why is the MIC technology so exciting?

How do we know?

How to program and optimize for MIC?

TACC

TACC — Texas Advanced Computing Center

- World-wide reputation for computational excellence
- Large clusters for compute and visualization
  - Ranger w/ 579 TFlops — Lonestar w/ 302 Tflops
  - Longhorn: 512 GPUs
- Large research projects

Intel® MIC Architecture

- Fascinating technology — **Inviting** programming models
- Tremendous potential for Scientific Computing
- Opens a road to Exascale computing with Intel® Xeon®

TACC + Intel + Dell + Academic Partners
Stampede Cluster in Q1 2013

- **~10 PFlops, 80% from MIC**

TACC

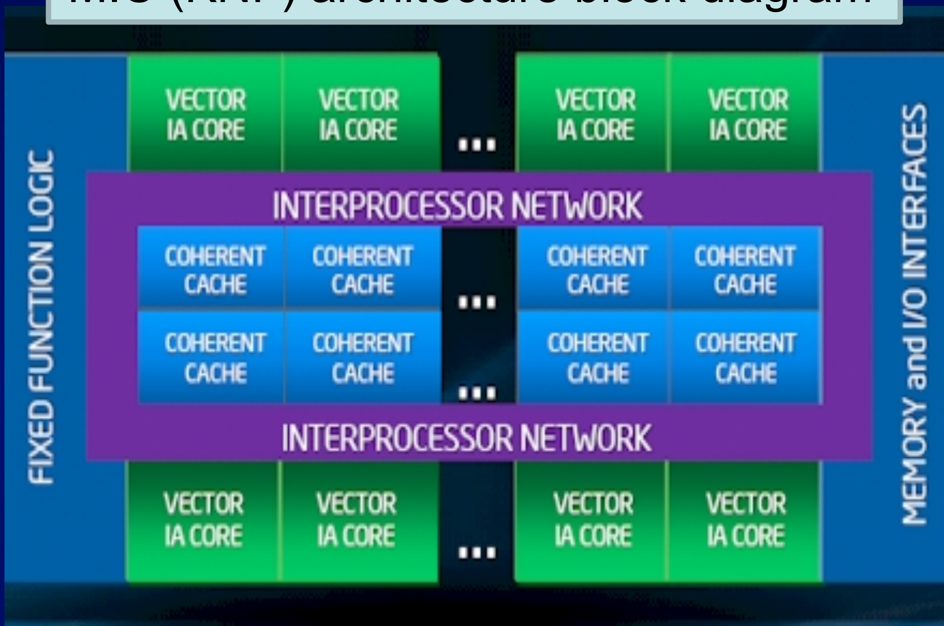
Programming and Optimizing for MIC?

How hard can it be?

MIC Architecture

- Many cores on the die
- L1 and L2 cache
- Bidirectional ring network
- Memory and PCIe connection

MIC (KNF) architecture block diagram



Knights Ferry SDP

- Up to 32 cores
- 1-2 GB of GDDR5 RAM
- 512-bit wide SIMD registers
- L1/L2 caches
- Multiple threads (up to 4) per core
- Slow operation in double precision

Knights Corner in Stampede

- 61 cores
- 8 GB of GDDR5 memory

What we at TACC like about MIC

(and we think that **you** will like this, too)

- Intel's® MIC is based on x86 technology
  - x86 cores w/ caches and cache coherency
  - SIMD instruction set
- Programming for MIC is similar to programming for CPUs
  - Familiar languages: C/C++ and Fortran
  - Familiar parallel programming models: OpenMP & MPI
  - MPI on host and on the coprocessor
  - Any code can run on MIC, not just kernels
- Optimizing for MIC is similar to optimizing for CPUs
  - Make use of existing knowledge!

Key elements of this talk highlighted!

Coprocessor vs. Accelerator

- Differences
  - Architecture: x86 vs. streaming processors

    coherent caches vs. shared memory and caches
  - HPC Programming model:

    extension to C++/C/Fortran vs. CUDA/OpenCL
    OpenCL support

  Threading/MPI:

    OpenMP and Multithreading vs. threads in hardware

    MPI on host and/or MIC vs. MPI on host only
  - Programming details

    offloaded regions vs. kernels
  - Support for any code: serial, scripting, etc.

    Yes No
- Native mode: Any code may be "offloaded" as a whole to the coprocessor

Adapting Scientific Code to MIC

- Today: Most scientific code for clusters
  - Languages: C/C++ and/or Fortran,
  - Communication: MPI
  - may be thread-based (Hybrid code: MPI & OpenMP),
  - may use external libraries (MKL, FFTW, etc.).

- With MIC on Stampede:
  - Languages: C/C++ and/or Fortran,
  - Communication: MPI
  - may run an MPI task on the MIC
    or may offload sections of the code to the MIC,
  - will be thread-based (Hybrid code: MPI & OpenMP),
  - may use external libraries (MKL),
    that automatically use MIC

TACC

Programming Models

Ready to use on day one!

- TBB's will be available to C++ programmers
- MKL will be available
  - Automatic offloading by compiler for some MKL features
- Cilk Plus
  - Useful for task-parallel programing (add-on to OpenMP)
  - May become available for Fortran users as well
- OpenMP
  - TACC expects that OpenMP will be the most interesting programming model for our HPC users

Execution Models

Questions:

Where to place the MPI tasks?

Which MPI task spawns the threads on the MIC?

Answers:

Two execution models come to mind:

Symmetric vs. "Offloaded"

TACC

MPI Task Placement and Communication

Symmetric setup: Easier

- MPI tasks on host and coprocessor
- ➢ Equal (symmetric) members of the MPI communicator
- Same code on host processor and MIC processor
- Communication between any MPI tasks through regular MPI calls

"Offloaded" setup: More involved

- MPI tasks on host only
- "Offload" directives added to OpenMP directives
- Communication between host and MIC through "offload" semantics

Symmetric Execution Model

- MPI tasks on host and coprocessor
- ➢ Equal (symmetric) members of the MPI communicator
- Same code on host processor and MIC processor
- Communication between any MPI tasks through regular MPI calls
  - ➢ Host ↔ Host
  - ➢ Host ↔ MIC
  - ➢ MIC ↔ MIC

TACC

"Offloaded" Execution Model

- MPI task execute on the host
- Directives "offload" OpenMP code sections to the MIC
- Communication between MPI tasks on hosts through MPI
- Communication between host and coprocessor through "offload" semantics
- Code modifications:
  - "Offload" directives inserted before OpenMP parallel regions

One executable (a.out) runs on host *and* coprocessor

Two Models: When to use Which? (1)

- Premise: Any code
  - contains parallel and serial sections
  - scales well if the serial sections are small / short
- Xeon vs. MIC Coprocessor
  - Xeon optimized for “any/irregular” workload
  - ➢ Executes serial sections faster
  - MIC, many cores optimized for “regular” workload
  - ➢ Executes parallel sections much faster
- Conclusion
  - Minimizing the time spent in serial sections is even more critical on MIC than on regular hosts

Two Models: When to use Which? (2)

Case 1:

- Serial code sections are very small
- Serial sections can be executed on MIC without substantial impact on overall performance

→ Use symmetric execution model

Case 2:

- Algorithm / implementation contains unavoidable large serial code sections
- Serial sections execution much better on a single Xeon core than on a single MIC core

→ Use "offloaded" execution model

MIC Programming with Offloading and OpenMP

- MIC specific pragma precedes OpenMP pragma
  - Fortran: `!dir$ omp offload target(mic) <…>`
  - C: `#pragma offload target(mic) <…>`
- Without optional keywords, all data transfer is handled by the compiler
- Example 1: Automatic data management
- Example 2: Manual data management
- Example 3: I/O from within offloaded region
  - Data can “stream” through the MIC; no need to leave the coprocessor to fetch new data
  - Also very helpful when debugging (print statements)
- Example 4: Offloading a subroutine and using MKL

TACC

Example 1

- 2-D array (`a`) is filled with data on the coprocessor
- Data management handled automatically by the compiler
  - Memory for (`a`) allocated on coprocessor
  - Private variables (`i, j, x`) are created
  - Result is copied back

```
program ex1    ! Fortran example
!$ use omp_lib                                ! OpenMP
integer                  :: n = 1024          ! Size
real, dimension(:,:), allocatable :: a ! Array
integer                  :: i, j              ! Index
real                     :: x                 ! Scalar

allocate(a(n,n))                           ! Allocation

!dir$ omp offload target(mic)              ! Offloading
!$omp parallel do shared(a,n), &           ! Par. region
  private(x, i, j), schedule(dynamic)
do j=1, n
  do i=j, n
    x = real(i + j); a(i,j) = x
  enddo
enddo
end program ex1
```

```
#include <omp.h>        /* C example */
#include <stdlib.h>
#include <stdio.h>
int main() {
  const int n = 1024; /* Size of the array */
  float    a[n][n];     /* Array            */
  int      i, j;        /* Index variables  */
  float    x;           /* Scalar           */

#pragma offload target(mic)
#pragma omp parallel for shared(a), \
        private(x), schedule(dynamic)
  for(i=0;i<n;i++) {
    for(j=i;j<n;j++) {
      x = (float)(i + j); a[i][j] = x;
    }
  }
}
```

Example 2

- Stencil update and Reduction with persistent data
- Data management by programmer
  - Copy in without deallocation: `in(a: free_if(0))`
  - First and second use without any data movement: `nocopy(a)`
  - Finally, copy out without allocation: `out(a: alloc_if(0))`

```
!!! Array a is a 2d array with (0:n+1,0:n+1)
elements

! Data transfer with allocation, no deallocation
!dir$ omp offload target(mic) in(a: free_if(0))
!$omp parallel
!$omp end parallel

! Offloading: no allocation and data transfer
!dir$ omp offload target(mic) nocopy(a)
!$omp parallel do shared(a)
do j=1, n
  do i=1, n
    a(i,j) = 0.25 * (a(i+1,j) + a(i-1,j) + &
                     a(i,j-1) + a(i,j+1))
  enddo
enddo

sum = 0. ! host code between offloaded regions
```

```
! Offloading: no allocation and data transfer
!dec$ omp offload target(mic) nocopy(a)
!$omp parallel do shared(a) reduction(+:sum)
do j=1, n
  do i=1, n
    sum = sum + a(i,j)
  enddo
enddo

! Data transfer with deallocation, no allocation
!dir$ omp offload target(mic) out(a: alloc_if(0))
!$omp parallel
!$omp end parallel
```

Example 3

- I/O from within offloaded region
- File opened/closed by one thread (omp single)
- Threads read from file (omp critical)
- Threads may read in parallel (not shown)
  - Parallel file system
  - Threads read different parts of file, stored on different targets

```
#pragma offload target(mic) /* Offloaded region */
#pragma omp parallel
{
#pragma omp single /* Open File */
{
  printf("Opening file in offloaded region\n");
  f1 = fopen("/var/tmp/mydata/list.dat","r");
}

#pragma omp for
for(i=1;i<n;i++) {
#pragma omp critical
  {
    fscanf(f1,"%f",&a[i]);
  }
  a[i] = sqrt(a[i]);
}

#pragma omp single
{
  printf("Closing file in offloaded region\n");
  fclose (f1);}}
```

Example 4

- Two routines **sgemm** (MKL) and **my_sgemm**
- Both called with offload directive
  - Explicit data movement used for **my_sgemm**
  - Input: **in(a, b)**
  - Output: **out(d)**
- Hand-written code (**my_sgemm**) carries special attribute to have routine compiled for the coprocessor

```
! Snippet from the Main Program
!dir$ attributes offload:mic :: sgemm

!dir$ offload target(mic) ! Offload to MIC
call sgemm('N','N',n,n,n,alpha,a,n,b,n,beta,c,n)

! Offload to the accelerator with explicit
!   clauses for the data movement
!dir$ offload target(mic) in(a,b) out(d)
call my_sgemm(d,a,b)
```

```
! Snippet from the Hand-coded subprogram
!dir$ attributes offload:mic :: my_sgemm
subroutine my_sgemm(d,a,b)
real, dimension(:,:) :: a, b, d
!$omp parallel do
do j=1, n
  do i=1, n
    d(i,j) = 0.
    do k=1, n
      d(i,j) = d(i,j) + a(i,k) * b(k,j)
    enddo
  enddo
enddo
end subroutine
```

Thank You!

Questions?