Xeon Phi programming on SuperMIC

Shaohao Chen

High performance computing @ Louisiana State University
Outline

- Intel Xeon Phi and its computing features
- Usage of Xeon Phi in HPC
- Xeon Phi programming
  - native mode
  - offloading
  - symmetric processing
Multi-core vs. Many-core

Xeon

Intel Many Integrated Core (MIC) Architecture

- 8 ~ 12 cores
- Single-core 2.5 ~ 3 GHz
- 256-bit vectors

Xeon Phi

- 61 cores (244 logical)
- Single-core ~ 1.2 GHz
- 512-bit vectors
Intel Xeon Phi coprocessor (accelerator)

(parameters for Xeon Phi 7120P)

- Add-on to CPU-based system
- PCI express (6.66 ~ 6.93 GB/s)
- IP-addressable
- 16 GB memory
- 61 x86 64-bit cores (244 threads)
- single-core 1.2 GHz
- 512-bit vector registers
- $1.208 \text{TeraFLOPS} = 61 \text{ cores} \times 1.238 \text{ GHz} \times 16 \text{ DP FLOPs/cycle/core}$

Current: Knight Corner (KNC)
Next (2016): Knight Landing (KNL)
Xeon Phi Computing Performance

Theoretical Maximums

(2S Intel® Xeon® processor E5-2670 & E5-2697v2 vs. Intel® Xeon Phi™ coprocessor)

**Single Precision**
- Up to 3.6x
- Higher is Better
- E5-2670
- E5-2697v2
- 3120PA
- 5110P
- 5110L
- 7120P

**Double Precision**
- Up to 3.6x
- Higher is Better
- E5-2670
- E5-2697v2
- 3120PA
- 5110P
- 5110L
- 7120P

**Memory Bandwidth**
- Up to 3.45x
- Higher is Better
- E5-2670
- E5-2697v2
- 3120PA
- 5110P
- 5110L
- 7120P

Source from Intel website
Synthetic Benchmark Summary (1 of 2)

**SGEMM (GF/s)**
- Higher is Better
- Up to 3.32x Higher

| Processor | Intel MKL
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ES-2670</td>
<td>670</td>
</tr>
<tr>
<td>ES-2697v2 (2x 2.6GHz, 8C, 130W)</td>
<td>1,059</td>
</tr>
<tr>
<td>3120P</td>
<td>1,722</td>
</tr>
<tr>
<td>5110P (60C, 1.053GHz, 225W)</td>
<td>1,741</td>
</tr>
<tr>
<td>5120D</td>
<td>1,742</td>
</tr>
<tr>
<td>7120P (61C, 1.238GHz, 300W)</td>
<td>2,221</td>
</tr>
<tr>
<td>7120D</td>
<td>2,225</td>
</tr>
</tbody>
</table>

**DGEMM (GF/s)**
- Higher is Better
- Up to 3.2x Higher

| Processor | Intel MKL
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ES-2670</td>
<td>332</td>
</tr>
<tr>
<td>ES-2697v2 (2x 2.6GHz, 8C, 130W)</td>
<td>548</td>
</tr>
<tr>
<td>3120P</td>
<td>818</td>
</tr>
<tr>
<td>5110P (60C, 1.053GHz, 225W)</td>
<td>837</td>
</tr>
<tr>
<td>5120D</td>
<td>837</td>
</tr>
<tr>
<td>7120P (61C, 1.238GHz, 300W)</td>
<td>1,064</td>
</tr>
<tr>
<td>7120D</td>
<td>1,067</td>
</tr>
</tbody>
</table>
Synthetic Benchmark Summary (2 of 2)

**Linpack** (GF/s)

<table>
<thead>
<tr>
<th>Using Intel MKL</th>
<th>Linpack(^1)</th>
<th>Higher is Better</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5-2670 (2x 2.6GHz, 8C, 115W)</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>E5-2697v2 (2x 2.7GHz, 12C, (57C, 300W))</td>
<td>543</td>
<td></td>
</tr>
<tr>
<td>3120P (1.1GHz, 60C, 1.053GHz, 60C, 1.053GHz, 60C, 1.23GHz, 60C, 1.23GHz, 245W)</td>
<td>715</td>
<td></td>
</tr>
<tr>
<td>5110P (2.2GHz, 225W)</td>
<td>769</td>
<td></td>
</tr>
<tr>
<td>5120D (1.0GHz, 245W)</td>
<td>767</td>
<td></td>
</tr>
<tr>
<td>7120P (1.0GHz, 300W)</td>
<td>999</td>
<td>Up to 3.0x Higher</td>
</tr>
<tr>
<td>7120D (1.0GHz, 300W)</td>
<td>1003</td>
<td></td>
</tr>
</tbody>
</table>

**STREAM Triad** (GB/s)

<table>
<thead>
<tr>
<th>Using Intel MKL</th>
<th>STREAM Triad</th>
<th>Higher is Better</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5-2670 (2x 2.6GHz, 8C, 115W)</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>E5-2697v2 (2x 2.7GHz, 12C, (57C, 300W))</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>3120P (1.1GHz, 60C, 1.053GHz, 60C, 1.053GHz, 60C, 1.23GHz, 60C, 1.23GHz, 245W)</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>5110P (2.2GHz, 225W)</td>
<td>165</td>
<td>Up to 2.3x Higher</td>
</tr>
<tr>
<td>5120D (1.0GHz, 245W)</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>7120P (1.0GHz, 300W)</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>7120D (1.0GHz, 300W)</td>
<td>178</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) Linpack is a benchmark for measuring the performance of floating-point operations in a computer system.
Intel® Xeon Phi™ Coprocessor vs. 2S Intel® Xeon® processor (Intel® MKL)

Using Intel® MKL High is Better

Intel® Xeon Phi™ Coprocessor 7120P vs.
2 Socket Intel® Xeon® processor (ES-2697v2)

Relative Performance (Normalized to 1.0 Baseline of a 2 socket Intel® Xeon® processor ES-2697v2)

Native = Benchmark run 100% on coprocessor. AO = Automatic Offload Function = Xeon + Xeon Phi together
<table>
<thead>
<tr>
<th>Segment</th>
<th>Customer</th>
<th>Application</th>
<th>Performance Increase(^1) vs. 2S Xeon*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>Acceleware</td>
<td>8(^{th}) order isotropic variable velocity</td>
<td>Up to 2.23x</td>
</tr>
<tr>
<td></td>
<td>Sinopec</td>
<td>Seismic Imaging</td>
<td>Up to 2.53x(^2)</td>
</tr>
<tr>
<td></td>
<td>CNPC (China Oil &amp; Gas)</td>
<td>GeoEast Pre-Stack Time Migration (Seismic)</td>
<td>Up to 3.54x(^2)</td>
</tr>
<tr>
<td>Financial Services</td>
<td>Financial Services</td>
<td>BlackScholes SP</td>
<td>Up to 7.5x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Monte Carlo SP</td>
<td>Up to 10.75x</td>
</tr>
<tr>
<td>Physics</td>
<td>Jefferson Labs</td>
<td>Lattice QCD</td>
<td>Up to 2.79x</td>
</tr>
<tr>
<td>Finite Element</td>
<td>Sandia Labs</td>
<td>miniFE (Finite Element Solver)</td>
<td>Up to 2x(^3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Up to 1.3x(^5)</td>
</tr>
<tr>
<td>Solid State Physics</td>
<td>ZIB (Zuse-Institut Berlin)</td>
<td>Ising 3D (Solid State Physics)</td>
<td>Up to 3.46x</td>
</tr>
<tr>
<td>Digital Content Creation/Video</td>
<td>Intel Labs</td>
<td>Ray Tracing (incoherent rays)</td>
<td>Up to 1.88x(^4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Video Transcoding</td>
<td>Up to 3.0x(^2)</td>
</tr>
<tr>
<td>Astronomy</td>
<td>CSIRO/ASKAP (Australia Astronomy)</td>
<td>tHogbom Clean (Astronomy image smear removal)</td>
<td>Up to 2.27x</td>
</tr>
<tr>
<td></td>
<td>TUM (Technische Universität München)</td>
<td>SG++ (Astronomy Adaptive Sparse Grids/Data Mining)</td>
<td>Up to 1.7x</td>
</tr>
<tr>
<td>Fluid Dynamics</td>
<td>AWE (Atomic Weapons Establishment - UK)</td>
<td>Cloverleaf (2D Structured Hydrodynamics)</td>
<td>1.77x</td>
</tr>
</tbody>
</table>
Intel Xeon Phi vs. Nvidia GPU

Disadvantages
◇ Less acceleration
◇ In terms of computing speed, one GPU beats one Xeon Phi for most cases currently.

Advantages
◇ X86 architecture
◇ IP-addressable
◇ Traditional parallelization (OpenMP, MPI)
◇ Easy programming, minor changes from your old codes
◇ Native and symmetric: no need to modify source codes, only changes in compiling and running codes.
◇ Offload: minor change of source code, similar to OpenACC but much less efforts than CUDA.
◇ New. Still a lot of room for improvement.
Usage of Xeon Phi in HPC

Statistics of accelerators in top 500 supercomputers (June 2014 list)

<table>
<thead>
<tr>
<th>Accelerator/CP Family</th>
<th>Count</th>
<th>System Share (%)</th>
<th>Rmax (GFlops)</th>
<th>Rpeak (GFlops)</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>436</td>
<td>87.2</td>
<td>179,221,078</td>
<td>252,023,035</td>
<td>15,384,521</td>
</tr>
<tr>
<td>Nvidia Kepler</td>
<td>23</td>
<td>4.6</td>
<td>35,133,206</td>
<td>52,529,949</td>
<td>960,416</td>
</tr>
<tr>
<td>Nvidia Fermi</td>
<td>20</td>
<td>4</td>
<td>9,813,752</td>
<td>20,123,473</td>
<td>750,056</td>
</tr>
<tr>
<td>Intel Xeon Phi</td>
<td>16</td>
<td>3.2</td>
<td>47,390,611</td>
<td>75,176,932</td>
<td>4,234,766</td>
</tr>
<tr>
<td>Hybrid</td>
<td>3</td>
<td>0.6</td>
<td>1,373,234</td>
<td>2,018,888</td>
<td>236,284</td>
</tr>
<tr>
<td>ATI Radeon</td>
<td>2</td>
<td>0.4</td>
<td>831,900</td>
<td>1,686,749</td>
<td>83,328</td>
</tr>
</tbody>
</table>
Number of supercomputers with Xeon Phi coprocessors

<table>
<thead>
<tr>
<th>List</th>
<th>Count</th>
<th>System Share (%)</th>
<th>Rmax (GFlops)</th>
<th>Rpeak (GFlops)</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun 2014</td>
<td>16</td>
<td>3.2</td>
<td>47,390,611</td>
<td>75,176,932</td>
<td>4,234,766</td>
</tr>
<tr>
<td>Nov 2013</td>
<td>12</td>
<td>2.4</td>
<td>45,244,135</td>
<td>72,197,351</td>
<td>4,079,172</td>
</tr>
<tr>
<td>Jun 2013</td>
<td>11</td>
<td>2.2</td>
<td>42,131,863</td>
<td>67,790,175</td>
<td>3,830,503</td>
</tr>
<tr>
<td>Nov 2012</td>
<td>7</td>
<td>1.4</td>
<td>4,302,784</td>
<td>6,309,356</td>
<td>337,301</td>
</tr>
<tr>
<td>Jun 2012</td>
<td>1</td>
<td>0.2</td>
<td>118,600</td>
<td>180,992</td>
<td>9,800</td>
</tr>
</tbody>
</table>
## Typical supercomputers with Xeon Phi coprocessors

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
</table>
| 1    | National Super Computer Center in Guangzhou (/site/50365) China     | Tianhe-2 (MilkyWay-2) -  
TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C  
2.200GHz, TH Express-2, Intel Xeon Phi 31S1P  
(/system/177999) NUDT                      | 3120000 | 33862.7       | 54902.4       | 17808      |
| 7    | Texas Advanced Computing Center/Univ. of Texas (/site/48958) United States | Stampede - PowerEdge C8220, Xeon E5-2680 8C  
2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P  
(/system/177931) Dell                       | 462462 | 5168.1        | 8520.1        | 4510       |
| 65   | Louisiana State University (/site/48279) United States               | SuperMIC - Dell C8220X Cluster, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, Intel Xeon Phi 7120P  
(/system/178423) Dell                        | 45866  | 557.0         | 925.1          | 370        |
SuperMIC @ LSU

- 360 Compute Nodes
  - Two 2.8GHz 10-Core Ivy Bridge-EP E5-2680 Xeon 64-bit Processors
  - Two Intel Xeon Phi 7120P Coprocessors
  - 64GB DDR3 1866MHz Ram
  - 500GB HD
  - 56 Gigabit/sec Infiniband network interface
  - 1 Gigabit Ethernet network interface

Ref: http://www.hpc.lsu.edu/resources/hpc/system.php?system=SuperMIC
A typical compute node on SuperMIC

- **Two Intel® Xeon® CPUs**
  - Host
  - 20 cores
  - 64 GB memory

- **One Intel® Xeon Phi™ coprocessor**
  - Mic0
  - 61 cores (244 logical)
  - 16 GB memory

- **Two Intel® Xeon Phi™ coprocessors**
  - Mic1
  - 61 cores (244 logical)
  - 16 GB memory

- **Theoretical maximum acceleration:**
  - Oen Xeon Phi / Two Xeons = 1208 GFLOPS / 448 GFLOPS = 2.7
  - (Two Xeons + Two Xeon Phis) / Two Xeons = (2*1208 + 448) GFLOPS / 448 GFLOPS = 6.4
Xeon Phi programming

- **Native mode**
  - vectorization performance
- **Offloading**
  - Explicit offload
  - MKL automatic offload
  - offload MPI-OpenMP hybrid codes
- **Symmetric processing**
  - run on one node
  - run on multi nodes
Getting started …

- **Window 1 (run jobs)**
  - `ssh username@smic.hpc.lsu.edu`  # login SuperMIC
  - `qsub -I -A allocation_name -l nodes=2:ppn=20,walltime=hh:mm:ss`  # interactive session

- **Window 2 or 3 (monitor performance)**
  - `ssh -X username@smic.hpc.lsu.edu`  # login SuperMIC with graphics
  - `ssh -X smic{number}`  # login the compute node with graphics
  - `micsmc & ( or micsmc-gui & or micsmc -a )`  # open Xeon phi monitor from the host
  - `ssh smic{number}p-mic0`  # login mic0
  - `top`  # monitor processes on Xeon Phi
I. Native mode

- **Computer codes** execute natively

- **An example** (vector_omp.c): vector addition, parallelized with OpenMP.
  - No change to normal CPU source codes.
  - Always compile codes on host. Compiler is not available on Xeon Phi.

- **Compilation**
  - `icc -O3 -openmp vector_omp.c -o vec_omp.cpu` # CPU binary
  - `icc -O3 -openmp -mmic vector_omp.c -o vec_omp.mic` # MIC binary
- execute CPU binary on the host
  - export LD_LIBRARY_PATH=/usr/local/compilers/Intel/composer_xe_2013.5.192/compiler/lib/intel64
  - export OMP_NUM_THREADS=20  # set OpenMP threads on host. Maximum is 20.
  - ./vec.omp.cpu  # run on the host

- execute MIC binary on Xeon Phi natively
  - ssh smic{number} p-mic0  # login mic0
  - export LD_LIBRARY_PATH=/usr/local/compilers/Intel/composer_xe_2013.5.192/compiler/lib/mic
  - export OMP_NUM_THREADS=244  # Set OpenMP threads on mic0. Maximum is 244.
  - ./vec.omp.mic  # Run natively on mic0
Exercise 1: Native run and affinity setting

i) Compile vector_omp.c with and without the flag `-mmic`, then execute the binaries on the host and on Xeon Phi respectively.

ii) Set up the affinity environment (e.g. export `KMP_AFFINITY=compact,granularity=fine,verbose`), then execute the MIC binary natively on Xeon Phi. Change “compact” to “scatter” or “balanced” then run it again. Observe the outputs.
Vectorization performance

- Compare performance with and without vectorization
  - An example (vector.c): a serial code for vector addition.
    - `icc -O3 -openmp -mmic vector.c -o vec.mic # vectorized by default`
    - `icc -O3 -openmp -mmic -no-vec vector.c -o novec.mic # no vectorization`
  - Vectorized code is around 10 times faster!

- Exercise 2: vectorization
  i) Compile `vector.c` with and without the flag `-no-vec`, then run the binaries and compare the computational time.
  ii) Compile `vector.c` with the flags `-mmic` and `-vec-report3`, then vary the vector report number from 1 to 7. Observe the outputs.
Summary for Native mode

- Add flag `-mmic` to create MIC binary files.
- ssh to MIC and execute MIC binary natively.
- Vectorization is critical.
- Monitor MIC performance with `micsmc`. 
II. Offloading

- A simple C code with explicit offload (off02block.c)

```c
int totalProcs;
int maxThreads;

#pragma offload target(mic:0)
{
    // begin offload block
    totalProcs = omp_get_num_procs();
    maxThreads = omp_get_max_threads();
}
    // end offload block

printf("total procs: %d\n", totalProcs);
printf("max threads: %d\n", maxThreads);
```
Explicit Offload: compilation and run

- **Compilation**
  - **Without** `-mmic`, the same as compiling normal CPU codes.
  - `icc -openmp name.c -o name.off  # C`
  - `ifort -openmp name.f90 -o name.off  # Fortran`

- **Execute offloading jobs from the host**
  - `export MIC_ENV_PREFIX=MIC  # necessary to set the prefix whenever execute mic-related jobs from the host.`
  - `export MIC_OMP_NUM_THREADS=240  # set OepnMP threads for Xeon Phi (The default is the maximum, that is 240, not 244. Leave one core with 4 threads to execute offloading.)`
  - `./name.off`
Exercise 3 (a): report offloading information

i) Compile off02block.c, then execute it from the host.

ii) Export the value of OFFLOAD_REPORT in the range of 1, 2 and 3. Then run it again and analyze the outputs.

iii) Compile off02block.c with the flag -opt-report-phase=offload. Observe the outputs.

Exercise 3 (b):

Do exercise 3 (a) with the Fortran code off02block.f90.
Offload an OpenMP region

- Assign values to a vector (C code: off03omp.c)

```c
double a[500000];  // placed on host
int i;  // placed on host

#pragma offload target(mic:0)  // auto pass i and a in and out of MIC
#pragma omp parallel for
for (i=0; i<500000; i++) {
    a[i] = (double)i;
}

printf("\n\tlast val = %f \n", a[500000-1]);  // output
```

- Exercise 4 (a): Compile and run off03omp.c. Report offloading information and analyze data transfer between host and MIC.
Assign values to a vector (Fortran code: off03omp.f90)

```fortran
integer, parameter :: N = 500000
real :: a(N)   ! placed on host
!dir$ offload target(mic:0)  ! auto pass i and a in and out of MIC
!$omp parallel do
    do i=1,N
        a(i) = real(i)
    end do
!$omp end parallel do
!$omp target
print*, "last val is ", a(N)   ! output
```

Exercise 4 (b): Compile and run off03omp.f90. Report offloading information and analyze data transfer between host and MIC.
Place valuables on MIC

- attribute decorations (C code: off05global.c)

```c
......
__attribute__(( target(mic:0) )) int myGlobalInt; // global valuable, available on MIC
int main( void ) {
......
int myLocalInt = 123; // local valuable, not available on MIC
__attribute__(( target(mic:0) )) int myStaticInt; // local valuable, available on MIC
#pragma offload target(mic:0){ // auto IN: myLocalInt; auto OUT: myGlobalInt, myStaticInt
    myGlobalInt = 2 * myLocalInt;
    myStaticInt = 2 * myGlobalInt;
}
......
}
```
module mymodvars

!dir$ attributes offload:mic :: mymoduleint

integer :: mymoduleint ! global valuable, available on MIC

end module mymodvars

program main

use mymodvars

implicit none

integer :: mylocalint = 123 ! local valuable, not available on MIC

integer, save :: mysaveint ! local valuable, available on MIC

!dir$ offload begin target(mic) ! auto IN: mylocalint; auto OUT: mymoduleint, mysaveint

mymoduleint = 2 * mylocalint

mysaveint = 2 * mymoduleint

!dir$ end offload

....... 

end program
Exercise 5 (a):declspec/attribute decorations

i) Compile and run off05global.c.

ii) Report offloading information and analyze data transfer between host and MIC.

iii) Replace all __attribute__(( target(mic:0) )) with __declspec(target(mic:0)), then compile and run it again.

iv) Remove all attribute/declspec decorations, then compile the code with the flag “-offload-attribute-target=mic”. Run it again.

Exercise 5 (b):

Do sections i, ii and iv of exercise 5 (a) with the Fortran code off05global.f90.
Control data transfer between host and MIC

- `in, out, inout` (C code: off06stack.c)

```
......

double a[100000], b[100000], c[100000], d[100000];

int i;

for ( i=0; i<100000; i++ ) {
    a[i] = 1.0;  b[i] = (double)(i);
}

#pragma offload target(mic:0) 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];
}

......
```
Manage MIC memory

- Associated with dynamically allocated memory on the host.
- `alloc_if, free_if` (C code: off07heap.c)

```
......
int i; int N = 5000000; double *a, *b;
a = ( double* ) memalign( 64, N*sizeof(double) );
b = ( double* ) memalign( 64, N*sizeof(double) );
for ( i=0; i<N; i++ ) { a[i] = (double)(i); }

#pragma offload target(mic:0) in( a : length(N) alloc_if(1) free_if(1) ), \
    out( b : length(N) alloc_if(1) free_if(1) ) // length(N) is required for dynamically allocated arrays
#pragma omp parallel for
    for ( i=0; i<N; i++ ) {
        b[i] = 2.0 * a[i];
    }
```

......
Exercise 6 (a): control data transfer

i) Compile and run off06stack.c and off07heap.c.

ii) Report offloading information and analyze data transfer between host and MIC in these cases.

Exercise 6 (b):

Do exercise 6 (a) with the Fortran codes off06stack.f90 and off07heap.f90.
Asynchronous offload

- wait, signal, offload_wait (C code: off08asynch.c)

......

int n = 123;

#pragma offload target(mic:0) signal( &tag ) { //record this offload event &tag
    printf("\n\tlogical cores on mic: %d\n\n", omp_get_num_procs()); // total MIC threads
    printf("\n\tmaximum threads on mic: %d\n\n", omp_get_max_threads()); // used MIC threads
    incrementSlowly(&n); // n increases 1 then sleep 2 seconds on MIC
}

...... // the host can do something else here, while MIC is busy.

#pragma offload_wait target(mic:0) wait( &tag ) { //Host does not execute the following codes until event &tag is finished.
    printf("\n\tlogical cores: %d\n\n", omp_get_num_procs()); // total host threads
    printf("\n\tmaximum threads: %d\n\n", omp_get_max_threads()); // used host threads
}

if ( n == 123 ) { printf("\n\nThe offload increment has NOT finished...\n"); } // n does not change without waiting
else { printf("\n\nThe offload increment DID finish successfully...\n"); } // n increases 1 after waiting
......
integer :: n = 123
!
! record this offload event &tag
!dir$ offload begin target(mic:0) signal(tag) ! n increases 1 then sleep 2 seconds on MIC

call incrementsslowly(n) ! the host can do something else here, while MIC is busy.
!
!dir$ end offload
!
!Host does not execute the following codes until event &tag is finished.
!
!dir$ offload_wait target(mic:0) wait(tag) ! Host does not execute the following codes until event &tag is finished.

p = omp_get_num_procs()
!
if (n.eq.123) then
    print *, "The offload increment has NOT finished... ", "n: ", n       // n does not change without waiting
else
    print *, "The offload increment DID finish successfully... ", "n: ", n     // n increases 1 after waiting
endif
!
Exercise 7 (a): asynchronous offload

i) Compile and run 0ff08asynch.c.

ii) Comment out the line with `wait`, then compile and run again. Does the value of `n` increase? Why?

iii) Change `offload_wait` to `offload`, then compile and run again. Observe the number of threads in the output. Is it changed? Why?

Exercise 7 (b):

Do sections i and ii of exercise 7 (a) with the Fortran code `off08asynch.f90`. For section iii, output the number of MIC threads instead of CPU threads.
Data-only offload

- Offload_transfer, nocopy (C code: off06stack.c)

```
......
a = ( double* ) memalign( 64, N*sizeof(double) );  // allocate alligned memory on host
b = ( double* ) memalign( 64, N*sizeof(double) );

#pragma offload_transfer target(mic:0) nocopy( a : length(N) alloc_if(1) free_if(0) ) \
   nocopy( b : length(N) alloc_if(1) free_if(0) ) signal( &tag1 )  // allocate memory on MIC

for ( i=0; i<N; i++ ) { a[i] = (double)(i); }  // assign value on host

// after tag1 is finished, copy a from host to MIC,
#pragma offload target(mic:0) wait( &tag1 ) in( a : length(N) alloc_if(0) free_if(0) ) \
   out( b : length(N) alloc_if(0) free_if(0) ) signal( &tag2 )
#pragma omp parallel for
for ( i=0; i<N; i++ ) { b[i] = 2.0 * a[i]; }  // calculate b on MIC
```
// (...... continued from the previous slide)

#pragma offload_transfer target(mic:0) wait( &tag2 ) \    // after tag2 is finished
        nocopy( a : length(N) alloc_if(0) free_if(1) ) \   // deallocate a on mic
        out( b : length(N) alloc_if(0) free_if(1) ) \   // copy b from mic to host, deallocate b on mic
        signal( &tag3 )

#pragma offload_wait target(mic:0) wait( &tag3 )    // wait until tag3 is finished
{
        printf( "\n\tlast a val = %f", a[N-1]);   // print values on the host
        printf( "\n\tt last b val = %f\n\n", b[N-1]);
}

......
Automatic offload with Intel MKL

- Intel Math Kernel Library (MKL):
  highly vectorized and threaded Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics functions.

- An example: Matrix product and addition, $C = \alpha A B + \beta C$

  ```c
  ...... 
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, p, alpha, A, p, B, n, beta, C, n); // Double-precision General Matrix Multiplication
  ...... 
  ```

  ```fortran
  ...... 
  ...... 
  ```
- No modification of the normal CPU source code!

- **Compilation** (the same for normal CPU code)
  - `icc -openmp -mkl ao_intel.c`
  - `ifort -openmp -mkl ao_intel.f`

- **Run auto-offloading jobs**
  - `export MKL_MIC_ENABLE=1` # enable auto offload, also set the prefix MIC_
  - `export OMP_NUM_THREADS=16` # set CPU threads
  - `export MIC_OMP_NUM_THREADS=240` # set MIC threads from the host
  - `export OFFLOAD_REPORT=2` # offload report level
  - `./ao_intel`

- Depending on the problem size and the current status of the devices, the MKL runtime will determine how to divide the work between the host CPU's and the Xeon Phi's
Exercise 8 (a): automatic offload with MKL

i) Compile and run ao_intel.c. Observe the usage of MICs on the “micsmc” monitor.

ii) Compare the computational time with and without automatic offload.

iii) Change the number of threads on the host and on the MICs. Observe the variation of computational time.

iv) Increase the problem size from small to large and observe the results. At what threshold(s) does MKL begin to use the MIC?

Exercise 8 (b):

Do exercise 8 (a) with the Fortran code ao_intel.f.
Using MPI and offload together

- MPI is required to run jobs on multi nodes.
- Offloading of MPI functions? No.
  - Calling MPI functions within an offload region is not supported.
- Offload OpenMP blocks in MPI-OpenMP hybrid codes? Yes!
- An example: Calculate the value of $\pi$.
  \[
  \int_0^1 \frac{4.0}{1 + x^2} \, dx = \pi
  \]

**Parallelization:** Distribute the integration grids into various MPI tasks, then spread every MPI task into various OpenMP threads.
MPI-OpenMP hybrid codes with offload

- Calculate pi (C code: pi_hybrid_off.c)

```c
......
MPI_Init( &argc, &argv );  // MPI functions
MPI_Comm_rank( MPI_COMM_WORLD, &mynrank );
MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
......
#pragma offload target (mic:mynrank) in(start_int,end_int) //Offload OepnMP block of each MPI task to one MIC
#pragma omp parallel private(iam,i,np){  // OpenMP block
    iam = omp_get_thread_num();
    np=omp_get_num_threads();
    printf("Thread %5d of %5d in MPI task %5d of %5d\n",iam,np,mynrank,nprocs);
    ......
}
......
```
Calculate pi (Fortran code: pi_hybrid_off.f90)

......
call mpi_init(ierr) // MPI functions
call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myrank,ierr)
......
!dir$ offload begin target (mic:myrank) in(start_int,end_int) //Offload OepnMP block of each MPI task to one MIC
    !$omp parallel private(iam,i,np) // OpenMP block
       iam = omp_get_thread_num()
       np=omp_get_num_threads()
       write(*,*) iam, myrank, np,nprocs
       ......
    !$omp end parallel
!dir$  end offload
......
- Use Intel MPI implementation (a better option than MVAPICH2)
  - module switch mvapich2/2.0/INTEL-14.0.2 impi/4.1.3.048/intel64
  - module load impi/4.1.3.048/intel64

- Compile
  - mpiicc -O3 -openmp pi_hybrid_off.c -o pi_hybrid.off
  - mpiifort -O3 -openmp pi_hybrid_off.f90 -o pi_hybrid.off

- Run with mpiexec.hydra
  - export OFFLOAD_REPORT=2 # level-2 offload report
  - export MIC_ENV_PREFIX=MIC # make prefix simple
  - export MIC_OMP_NUM_THREADS=240 # number of threads on MIC
  - export MIC_KMP_AFFINITY=scatter # affinity type on MIC
  - mpiexec.hydra -n 2 -machinefile nodefile ./pi_hybrid.off # specify node names in node file
Exercise 9 (a): Offload OpenMP blocks in MPI-OpenMP hybrid codes

i) Compile \texttt{pi\_hybrid\_off.c}, then run it on one node and two nodes respectively, with two MPI tasks per node. Observe the usage of MICs on the micsmc monitor.

ii) Change the number of threads on the MICs. Observe the variation of computational time.

iii) Change the number of MPI tasks to 1 per node, run it again. How many MICs on each node are utilized now?

iv) Compare the computational time of the following cases: 1) without offloading; 2) offload to one MIC; 3) offload to two MICs.

Exercise 9 (b):

Do exercise 9 (a) with the Fortran code \texttt{pi\_hybrid\_off.f90}.
Summary for offloading

- Explicitly offload blocks by adding lines started with `#pragma offload` or `!dir$ offload` in C or Fortran source codes respectively.
- Control data transfer with `in`, `out` and `inout`.
- Place valuables on MIC with `attribute` or `declspec` decorations.
- Use `wait` and `offload_wait` for asynchronous offload.
- Use `offload_transfer` for data-only offload.
- Auto offload MKL functions by setting `MKL_MIC_ENABLE=1`.
- Offload OpenMP blocks in MPI-OpenMP hybrid codes.
III. Symmetric processing

- Distribute MPI tasks “symmetrically” on both CPUs and MICs.
Compilation

- **Use Intel MPI implementation** (the only working one)
  - module switch mvapich2/2.0/INTEL-14.0.2 impi/4.1.3.048/intel64
  - module load impi/4.1.3.048/intel64

- **Create CPU and MIC binaries separately**
  - mpiicc -openmp name.c -o name.cpu # CPU binary, C code
  - mpiicc -openmp -mmic name.c -o name.mic # MIC binary, C code
  - mpiifort -openmp name.f90 -o name.cpu # CPU binary, Fortran code
  - mpiifort -openmp -mmic name.f90 -o name.mic # MIC binary, Fortran code

- The flag -openmp is unnecessary for a pure MPI job.
- **There is no change from normal CPU codes!**
Run jobs with mpiexec.hydra

#!/bin/bash
source setup_mpiMic.sh  # set up environments for mpiexec.hydra
export TASKS_PER_HOST=2  # number of MPI tasks per host
export THREADS_HOST=10  # number of OpenMP threads spawned by each task on the host
export TASKS_PER_MIC=3  # number of MPI tasks per MIC
export THREADS_MIC=80  # number of OpenMP threads spawned by each task on the MIC
export CPU_ENV="-env OMP_NUM_THREADS $THREADS_HOST"  # CPU run-time environments
export MIC_ENV="-env OMP_NUM_THREADS $THREADS_MIC -env LD_LIBRARY_PATH \ $MIC_LD_LIBRARY_PATH"  # MIC run-time environments
mpiexec.hydra \  # a command provided by the Intel MPI
- n $TASKS_PER_HOST -host smic022 $CPU_ENV ./name.cpu : \  # run on CPU
- n $TASKS_PER_MIC -host smic022p-mic0 $MIC_ENV ./name.mic : \  # run on mic0
- n $TASKS_PER_MIC -host smic022p-mic1 $MIC_ENV ./name.mic  # run on mic1
Exercise 10: run jobs with mpiexec.hydra

i) Compile pi_hybrid.c or pi_hybrid.f90, then run it with mpiexec.hydra on one compute node. Observe the usage of MICs on the micsmc monitor.

ii) Vary the numbers of MPI tasks and OpenMP threads. Find out the best combination of them so that the computational time is the shortest.

iii) Compare the computational time of the following cases: 1) use only CPU; 2) use only one MIC; 3) use CPU and one MIC; 4) use CPU and two MICs.
- **Number of MPI tasks on MIC**
  - The theoretical maximum is 61, which is equal to the number of cores on MIC.
  - The practical number should be much less than 61 due to the MIC-memory (16 GB) bottleneck!

- **An issue of using mpiexec.hydra**
  
The command lines become very messy if many nodes are utilized.
Run jobs with micrun.sym

- micrun.sym is a bash script to run symmetric jobs on SuperMIC.
  - Automatically obtains the target names, sets up the environments and constructs the complicated command lines.
  - Easy for running heavy jobs with many nodes.

- Usage of micrun.sym:
  - `micrun.sym -c /path/to/name.cpu -m /path/to/name.mic`

- Exercise 11: run jobs with micrun.sym
  - Redo exercise 10 on four compute nodes instead of one, using `micrun.sym` instead of `mpiexec.hydra`.
A PBS batch script using micrun.sym with Torque/Moab job scheduler

#!/bin/bash

#PBS -q workq
#PBS -A your_allocation
#PBS -l walltime=01:30:00
#PBS -l nodes=4:ppn=20
#PBS -V

......

module load impi/4.1.3.048/intel64    # load Intel MPI
export TASKS_PER_HOST=20    # number of MPI tasks per host
export THREADS_HOST=1      # number of OpenMP threads spawned by each task on the host
export TASKS_PER_MIC=30    # number of MPI tasks per MIC
export THREADS_MIC=1       # number of OpenMP threads spawned by each task on the MIC

micrun.sym -c /path/to/name.cpu -m /path/to/name.mic    # run with micrun.sym
Summary for symmetric processing

- Use Intel MPI (impi) implementation.
- Create CPU and MIC binaries with and without -mmic respectively.
- Run symmetric jobs on few nodes with mpiexec.hydra.
- Run symmetric jobs on many nodes with micrun.sym.
- Balance works on CPU and MIC to obtain the best performance.
Optimization, debug and profile

- Optimization is required for further acceleration!

- Debug and profile:
  - Intel VTune Amplifier
  - Intel Trace Analyzer and Collector
Final remarks

“Well..., I hear lots of stuffs from this training, but where should I start to accelerate my codes on Xeon Phi?”

- **MKL**
  - If your code employs MKL, congratulations! Some MKL functions are automatically offloaded to Xeon Phi.

- **Non-MKL**: If your code is .......
  - parallel with OpenMP, explicitly offload the OpenMP blocks to Xeon Phi.
  - parallel with pure MPI, run it symmetrically on both CPUs and Xeon Phis.
  - parallel with hybrid MPI and OpenMP, either explicitly offload the OpenMP blocks to Xeon Phi or run it symmetrically on both CPUs and Xeon Phis.
  - serial, most likely it becomes slower, because the frequency of a Xeon Phi core is much lower than that of a CPU core.
References