CSE 746 - Parallel and High Performance Computing Lecture 11 - Intel Phi

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
ppomorsk@sharcnet.ca
http://ppomorsk.sharcnet.ca/

SSH key setup

- need SSH key with forwarding enabled to access Intel Phi cards on SHARCNET
- SSH key will allow you to log in without password
- ssh-keygen -t dsa
- save output to default file if one does not already exist
- pick password (possible to leave blank)
- log in to SHARCNET using your password and add the contents of the public key file (.pub) to ~/.ssh/authorized_keys
- change permissions with chmod 600 ~/.ssh/authorized_keys

SSH key forwarding

- want to be able to go from login node to one of the compute nodes without password
- SSH key forwarding permits this
- ssh -A
- Forwarding needs some setup
- on Mac:
  edit /etc/ssh_config and add

  Host *
  ForwardAgent yes
- on Linux, add this to $HOME/.ssh/config
- run ssh-agent, ssh-add ?

Intel Xeon Phi

- uses Intel MIC (Many Integrated Core) architecture
- came out of cancelled Intel project to make a GPGPU (Larrabee)
- Knights Ferry - prototype , released 2010
- Knights Corner, first generations **(current)**, released starting 2012
- Knights Landing - second generation MIC, should be available in second half of 2015, will be available both in coprocessor format and in host format
- Knights Hill - third generation MIC, available around 2017

References

- Book: Parallel Programming and Optimization with Intel Xeon Phi Coprocessors
  http://www.colfax-intl.com/nd/xeonphi/book.aspx
  Price: $50
  Some of the exercises today come from that book
- Developer's QuickStart Guide
  https://software.intel.com/sites/default/files/managed/ee/4e/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf
- Developer zone
  https://software.intel.com/en-us/mic-developer

Intel Xeon Phi numbering

- first generation models (only ones currently out)
  3100 series - price-optimal (about $2000)
  5100 series - best performance per watt (about $2500)
  7100 series - top performance (about $4000)
- all about 1000 TFLOP/s
- memory 6 GB, 8 GB, 16 GB

Intel Xeon Phi

- models with 57, 60 and 61 cores
- each core based on Intel Pentium processor
- takes instructions from 4 hardware threads, hence uses 4 way hyper-threading, hence 60 physical cores will present themselves to the programmer as 240 logical cores
- does not support previous Intel SIMD instructions (SSE etc.)
- instead it has its own, new vector instructions set to utilize a dedicated 512-bit wide vector floating point unit (VPU) provided for each core. This holds 16 SP floats, 8 DP floats, 16 32-bit integer, and can operate on them in one cycle.
- ring topology for memory interconnect, next generations will use grid topology

Intel Xeon Phi advantages

- x86 compatible, though not actually x86, so needs a different executable compiled specifically for it, and will not run a standard x86 executable
- can use existing approaches and tools,
- can maintain common code base
- writing good code for Intel Phi is largely equivalent to writing good code for the CPU

Intel Phi

- Tianhe-2 , the #1 supercomputer on last Top500 list (November, 2014) uses Intel Phi
- of the 500 systems on the list, 75 use accelerators, of which 50 use NVIDIA GPUs, 25 use Intel Xeon Phi (MIC)
- Calcul Quebec has a cluster with 100 Intel Phis, installed on 50 nodes
  http://www.calculquebec.ca/en/resources/compute-servers/guillimin

  anyone with SHARCNET account can get access to it

Intel Phi performance

- conventional Intel Xeon
  2 sockets, 8 processors each, running at 3.4 GHz, so in total 54.4 GHz across all cores
- Intel Phi
  60 cores, each 1 GHz , so in total 60 GHz across all cores
- Intel Phi only about 10% better in raw cycles
- BUT Intel Phi has 512 bit SIMD registers versus 256 SIMD registers on conventional Xeon, which permits twice the work per cycle
- other architecture improvements like high theoretical bandwidth
- to get speed up, must use all cores (just like on GPU), plus code must be **vectorized** to take advantage of SIMD registers

Intel Phi performance

- memory access on Phi in theory up to 384 GB/s, in practice about half that, so it's about 3 times better than conventional CPU in that area
- memory latency better on conventional CPU because of better caches
- Hence Intel Phi will be superior to conventional CPU only for:
  case 1: compute bound problems where memory access is not a constraint
  case 2: applications with streamlined memory accesses, limited by memory bandwidth and not by latency

Intel Phi parallelism

- achieved via conventional OpenMP, MPI and OpenCL
- however, pure MPI problematic distributed memory approach problematic since memory is limited
  - on 8 GB Intel Phi have only 133 MB of memory per process for 60 MPI processes (one per physical core), just one quarter that for 240 MPI processes (one for each hyper-thread)
- mix of OpenMP and MPI probably the best approach

Intel Phi on SHARCNET

- just one node of goblin, node 49 (gb49)
- that node has 2 Intel Phi 5100 series, each 8 GB RAM
- Intel Phi programs can be compiled only on that node
- gb49 does not have standard modules, so before compiling do:
  source /opt/sharcnet/intel/15.0.1/bin/compilervars.sh intel64
  source /opt/sharcnet/intel/15.0.1/mkl/bin/mklvars.sh intel64
  export INTEL_LICENSE_FILE=/opt/sharcnet/intel/15.0.1/license/intel.lic
- need to log into goblin cluster with key forwarding enabled
  ssh -A your_username@goblin.sharcnet.ca
- from goblin, can log into the Intel Phi card with
  ssh gb49-mic0.goblin.sharcnet
  ssh gb49-mic1.goblin.sharcnet

Checking for Intel Phi presence

- on host
  lspci | grep -i “co-processor”
- when actually logged into Phi
  cat /proc/cpuinfo
- checking is important since program might just run on the CPU if Intel Phi is not found. This of course makes coding for the Phi quite easy as in some cases you don’t actually need a Phi to write and test code

Intel Phi modes

- explicit offload mode
  - code starts running on host, offloads some functions to the coprocessor (just like kernels in CUDA)
  - there is a memory bottleneck between host and coprocessor (as in GPU)
- native mode
  log into the coprocessor and run on it like you would on a conventional CPU
  note: code still has to be compiled on host, as the operating system on the coprocessor is limited and there is no compiler
- compile for MIC architecture (Intel Phi) with:
  icc -mmic hello.c -o hello_mic.x

"Hello world" on Intel Phi in native mode

- simple hello world program, will run both on conventional CPU and Intel Phi, but needs -mmic flag to be compiled for MIC architecture

```
#include <stdio.h>
#include <unistd.h>
int main(){
  printf("Hello world! I have %ld logical cores.\n",
  sysconf(_SC_NPROCESSORS_ONLN ));
}
```

To check if really running on MIC

- in some cases, if MIC not available, program will just run on conventional processor
- to tell where your program is running, use code like this:

```
#ifdef __MIC__
printf("running on MIC \n");
#else
printf("not running on MIC, offload to coprocessor failed \n");
#endif
```

Native mode programs

- employ conventional OpenMP, MPI, OpenCL etc.
- code written does not fundamentally differ from code written for conventional CPU
- in this lecture will focus on the other approach, the explicit offload model
- this model is best for combining the strengths of the conventional CPU and Intel Phi

"Hello world" on Intel Phi -explicit offload mode

- we want the print to happen on the Phi
- needs fflush(0); to ensure output appears before program ends
- processor output might still appear after the host output
- compile with icc, no special flags needed

```
#include <stdio.h>
#include <unistd.h>
int main(){

#pragma offload target(mic)
        {
                printf("Hello, this is the coprocessor. \n");
                fflush(0);
        }
        printf("Goodbye from host. \n");
}
```

#pragma offload target(mic)

- offloads the code in the block that follows to the Intel Phi coprocessor
- if coprocessor not present, code will generate an error
- if multiple processors present, can target specific ones target(mic:0)

Offloading functions

- functions which will be offloaded coprocessor need to be declared as such (just like kernels in CUDA)

```
…
__attribute__((target(mic))) void MyFunction() {
…
}
…
#pragma offload target(mic)
{
MyFunction();
}
```

Offloading diagnostics

- export OFFLOAD_REPORT=2

```
[ppomorsk@gb49:~/edu_stuff/CSE746_lec11_2015/hello] ./a.out
main function not running on MIC
[Offload] [MIC 0] [File]                    hello_offload.c
[Offload] [MIC 0] [Line]                    28
[Offload] [MIC 0] [Tag]                     Tag 0
[Offload] [HOST]  [Tag 0] [CPU Time]        0.387177(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data]   0 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000165(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data]   0 (bytes)
```

Transferring data to coprocessor

- static arrays of known size transferred automatically

```
void MyFunction() {
const int N = 1000;
int data[N];
#pragma offload target(mic)
{
int i;
for (i = 0; i < N; i++)
data[i] = 0; }
}
```

Transferring data to coprocessor

- dynamically allocated arrays need explicit transfer
- transfer possibilities: in,out,inout,nocopy (assumes data already present)

```
void MyFunction(const int N, int* data) {
#pragma offload target(mic) inout(data: length(N))
{
int i;
for (i = 0; i < N; i++)
data[i] = 0; }
}
```

Transferring data to coprocessor

- transferring global and static variable

```
int* __attribute__((target(mic))) data;
void MyFunction() {
static __attribute__((target(mic))) int N = 1000;
#pragma offload target(mic) inout(data: length(N))
{
int i;
for (i = 0; i < N; i++)
data[i] = 0; }
}
```

Transferring data to coprocessor

- can transfer data without any computation

```
#pragma offload target(mic) in(data: length(N))
{}

//same as

#pragma offload_transfer target(mic) in(data: length(N))
```

Exercise 1a

- write a program to add integer vectors A and B, store result in C
- store vectors as static arrays, with size known as compile time
- initialize A,B with some arbitrary values on host, compute CCheck array on host to check the coprocessor calculation
- offload A+B=C loop to MIC
- check C against CCheck on host

Exercise 1b

- code from 1a will fail once vectors increase past certain size
- rewrite code so that vectors are dynamically allocated at runtime

Persistent data

- can add allocate/free clauses
- alloc_if(b) - allocate if b true
- free_if(b) - free if b true

```
CreatePersistentData(N, persistent);
#pragma offload_transfer target(mic:0)  \
        in(persistent : length(N) alloc_if(1) free_if(0) )
for (int iter = 0; iter < nIterations; iter++) { SetupDataset(iter, dataset);
#pragma offload target(mic:0)  \
in  (dataset : length(N) alloc_if(iter==0) free_if(iter==nIterations-1) )  \
out  (results : length(N) alloc_if(iter==0) free_if(iter==nIterations-1) ) \
nocopy (persistent : length(N) alloc_if(0) free_if(iter==nIterations-1) )
  {
    Compute_results(N, dataset, results, persistent);
}
  ProcessResults(N, results);
}
```

Exercise 2a

- write a C++ code that performs serial matrix-vector multiplication on host, use double precision array of floats allocated on stack
  $A \; x \; b = c$
  where A is a m x n matrix
- for simplicity, initialize A to 1.0/n, and b to 1.0, so that result vector just has values 1.0
- Use Cilk Plus array notation to initialize array, eg:
  A[0:n*m]=1.0/(double)n;

```
for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
        c[i] += A[i*n+j] * b[j];
```

Exercise 2b

- modify code so that the multiplication loop is offloaded to coprocessor
- afterwards verify on host that result is correct
- test for maximum problem size

```
for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
        c[i] += A[i*n+j] * b[j];
```

Exercise 2c

- change allocation, keep b on the stack, but put A on the heap (use malloc)
- test for maximum problem size

```
for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
        c[i] += A[i*n+j] * b[j];
```

Exercise 2d

- improve the code so it can work on multiple vectors
- make the multiplication iterative, where at each iteration the vector b is
  b[:] = (double) iter;
- make sure the A matrix is transferred only once, by using
  #pragma offload_transfer

Exercise 2e

- modify result 2d so that only a single #pragma offload target(mic0) in directive accomplishes the same thing

Asynchronous operations

- it is possible to overlap host and coprocessor calculations
- use signal() / wait() pair in pragmas

```
#pragma     … signal(A)
{
coprocessor code
}

//host code which will execute asynchronously follows

#pragma … wait(A)   // could be eg. offload_transfer
// code proceeds past this point only once pragma assigned signal A finished
// things are synchronized

// host code here can use results from A
```

Exercise 2f

- modify code so that offloaded matrix-vector multiplication is performed asynchronously, while the same verification calculation is performed on host
- compare the two results on host afterwards