Parallel Programming and Optimization with Intel Xeon Phi Coprocessors

Introduction

Vadim Karpusenko, PhD

Colfax Workshop

Chapters 1-2 – September 20, 2013
Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

Handbook on the Development and Optimization of Parallel Applications for Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors

© Colfax International, 2013

It all comes down to PARALLEL PROGRAMMING! (applicable to processors and Intel® Xeon Phi™ coprocessors both)

Forward, Preface
Chapters:
1. Introduction
2. High Performance Closed Track Test Drive!
3. A Friendly Country Road Race
4. Driving Around Town: Optimizing A Real-World Code Example
5. Lots of Data (Vectors)
6. Lots of Tasks (not Threads)
7. Offload
8. Coprocessor Architecture
9. Coprocessor System Software
10. Linux on the Coprocessor
11. Math Library
12. MPI
13. Profiling and Timing
14. Summary
Glossary, Index

Learn more about this book: lotsofcores.com

This book belongs on the bookshelf of every HPC professional. Not only does it successfully and accessibly teach us how to use and obtain high performance on the Intel MIC architecture, it is about much more than that. It takes us back to the universal fundamentals of high-performance computing including how to think and reason about the performance of algorithms mapped to modern architectures, and it puts into your hands powerful tools that will be useful for years to come.
—Robert J. Harrison
Institute for Advanced Computational Science, Stony Brook University

Intel® Xeon Phi™ Coprocessor High Performance Programming
Jim Jeffers, James Reinders
Available since February 2013.

Intel® Xeon Phi™ Coprocessor High Performance Programming, Jim Jeffers, James Reinders, (c) 2013, publisher: Morgan Kaufmann

© 2013, James Reinders & Jim Jeffers, book image used with permission
Workshop Road Map

1. Introduction
   - Intel Xeon Phi architecture overview, Technical specifications
   - Three Layers of Parallelism
   - Heterogeneous Computing

2. Programming Models for Intel Xeon Phi Applications
   - Native applications for coprocessors and MPI
   - Explicit Offload programming model
   - Using multiple coprocessors and heterogeneous clustering
Workshop Road Map

Expressing Parallelism

- Vectorization (Single Instruction Multiple Data (SIMD) parallelism) and automatic vectorization
- Thread parallelism with OpenMP and Intel Cilk Plus
- Distributed-memory parallelization with Message Passing Interface (MPI)
Course Road Map

Optimizing Applications for Intel Xeon Phi Product Family

- Delving deeper into the topics of parallel programming
- Example-based optimization advice
- Scalar optimizations
- Improving data structures for streaming, unit-stride, local memory access
- Guiding automatic vectorization with compiler hints
- Reducing synchronization in parallel algorithms by the use of reduction
- Avoiding false sharing and racing conditions
- Increasing arithmetic intensity and reducing cache misses by loop blocking and recursion
- Exposing the full scope of available parallelism
- Scheduling practices for load balancing across cores and MPI processes
§1. Introduction

Intel Xeon Phi Coprocessors and the Intel MIC Architecture
Intel Xeon Phi Coprocessors and the MIC Architecture

- Multi-core Intel Xeon processor
  - ~ 8 physical cores (16 logical) ~ 3GHz
  - 256-bit vector registers

- Many-core Intel Xeon Phi coprocessor
  - ~ 60 cores (240) ~ 1GHz
  - 512-bit vector registers
Intel Xeon Phi Coprocessors and the MIC Architecture

- Add-on to CPU-based systems (PCIe interface)
- High Power efficiency
- ~ 1 TFLOP/s in DP
- Heterogeneous clustering

For highly parallel applications which reach the scaling limits on Intel Xeon processors
Intel Xeon Phi Coprocessors and the MIC Architecture

Colfax CXP9000 with 8 Intel Xeon Phi coprocessors
N-body simulation on...

- Two Intel® Xeon® CPUs
- One Intel® Xeon Phi™ coprocessor
- Two Intel® Xeon Phi™ coprocessors
MIC Architecture: Developer’s Perspective

In common with Intel Xeon CPUs:
- x86 architecture
- C, C++ and Fortran
- Traditional parallelization libraries (OpenMP, MPI)
- Similar optimization methods

- PCIe bus connection
- IP-addressable
- own Linux µOS
- 8 GB cached GDDR5 (5110P)
- 60 x86 64-bit cores (5110P)
- 512-bit SIMD vector registers
- Performance Monitor Unit (PMU) on each core
- 4-way hyper-threading
Core Topology

- Instruction Decode
- Scalar Unit
- Scalar Registers
- L1 Cache (Data and Instruction)
- L2 Cache
- Interprocessor Network
- Vector Unit
- Vector Registers
- 4 Threads per Core
  - 64-bit
  - 512-wide
  - In Order
  - Specialized Instructions (new encoding)
  - HW transcendental
- Fully Coherent
  - L2 Hardware Prefetching
- Ring interconnect
  - 32 KB per core
  - 512 KB Slice per Core – Fast Access to Local Copy
The caches are 8-way associative, fully coherent with the LRU (Least Recently Used) replacement policy.

<table>
<thead>
<tr>
<th>Cache line size</th>
<th>64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 size</td>
<td>32KB data, 32KB code</td>
</tr>
<tr>
<td>L1 latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L2 size</td>
<td>512KB</td>
</tr>
<tr>
<td>L2 ways</td>
<td>8</td>
</tr>
<tr>
<td>L2 latency</td>
<td>11 cycles</td>
</tr>
<tr>
<td>Memory → L2 prefetching</td>
<td>hardware and software</td>
</tr>
<tr>
<td>L2 → L1 prefetching</td>
<td>software only</td>
</tr>
<tr>
<td>Translation Lookaside Buffer (TLB) coverage options (L1, data)</td>
<td>64 pages of size 4KB (256KB coverage)</td>
</tr>
<tr>
<td></td>
<td>8 pages of size 2MB (16MB coverage)</td>
</tr>
</tbody>
</table>
PCIe Bandwidth Considerations

When data must be sent from host to coprocessor, communication overhead must be considered:

- PCIe bandwidth: 6 GB/s, theoretical max arithmetic performance 1 TFLOP/s, practical memory bandwidth 200 GB/s
- Offload justified if algorithm performs $\gg 1000$ operations per offloaded word
- Algorithms with strong arithmetic complexity scaling (e.g., $O(n^2)$) likely less impacted by communication than algorithms with weak scaling (e.g., $O(n)$, $O(n \log n)$) for large problems
Three Layers of Parallelism

- Instruction Pool
- Data Pool
- SIMD
  - Vector Unit
  - PU
  - PU
  - PU
  - PU
Three Layers of Parallelism
Three Layers of Parallelism

- Instruction Pool
- Data Pool
- SIMD Vector Unit

Compute Node 1
- Host CPUs
- Xeon Phi coprocessor
- Xeon Phi coprocessor

MPI
Features of the IMCI Instruction Set

Intel IMCI is the instruction set supported by Intel Xeon Phi corp.

- **512-bit wide registers**
  - can pack up to eight 64-bit elements (long int, double)
  - up to sixteen 32-bit elements (int, float)

- **Arithmetic Instructions**
  - Addition, subtraction and multiplication
  - Fused Multiply-Add instruction (FMA)
  - Division and reciprocal calculation;
  - Error function, inverse error function;
  - Exponential functions (natural, base 2 and base 10) and the power function.
  - Logarithms (natural, base 2 and base 10).
  - Square root, inverse square root, hypothenuse value and cubic root;
  - Trigonometric functions (sin, cos, tan, sinh, cosh, tanh, asin, acos, atan);
  - Rounding functions
Features of the IMCI Instruction Set

- Initialization, Load and Store, Gather and Scatter
- Comparison
- Conversion and type cast
- Bitwise instructions: NOT, AND, OR, XOR, XAND
- Reduction and minimum/maximum instructions
- Vector mask instructions
- Scalar instructions
- Swizzle and permute
Ease of use

Threaded Options

- Intel® Math Kernel Library API*
- Intel® Threading Building Blocks Intel® Cilk™ Plus
- OpenMP*
- Pthreads*

Vector Options

- Intel® Math Kernel Library
- Array Notation: Intel® Cilk™ Plus
- Auto vectorization
- Semi-auto vectorization: #pragma (vector, ivdep, simd)
- OpenCL*
- C/C++ Vector Classes (F32vec16, F64vec8)
Two Levels of Parallelism

1. Scalar & Single-thread
2. Vector & Single-thread
3. Scalar & Multi-threaded
4. Vector & Multi-threaded

More Parallel

More Performance

- Intel Xeon Phi
- Intel Xeon
### Linux Host

**User-level code**
- Host-side offload application
  - User code
  - SSH
  - Offload libraries, user-level driver, user-accessible APIs and libraries

**System-level code**
- Linux OS
- PCIe Bus

### Intel® Xeon Phi™ coprocessor

**User-level code**
- Target-side “native” application
  - User code
  - Standard OS libraries plus any 3rd-party or Intel libraries

**System-level code**
- Intel® Xeon Phi™ coprocessor communication and application-launching support
- PCIe Bus
- Linux uOS
Heterogeneous Computing

- Xeon - Multi-Core Centric
- MIC - Many-Core Centric

**Breadth**

- Multi-Core Hosted
  - General serial and parallel computing
- Offload
  - Code with highly-parallel phases
- Symmetric
  - Codes with balanced needs
- Many Core Hosted
  - Highly-parallel codes
Interacting with Intel Xeon Phi coprocessors

```
user@host% lspci | grep -i "co-processor"
06:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)
82:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)

user@host% sudo service mpss status
mpss is running

user@host% cat /etc/hosts | grep mic
172.31.1.1  host-mic0  mic0
172.31.2.1  host-mic1  mic1

user@host% ssh mic0
user@mic0% cat /proc/cpuinfo | grep proc | tail -n 3
processor : 237
processor : 238
processor : 239

user@mic0% ls /
amplxe  dev  home  lib64  oldroot  proc  sbin  sys  usr
bin  etc  lib  linuxrc  opt  root  sep3.10  tmp  var
```
mic* Utilities at /opt/intel/mic/bin/

**micinfo** a system information query tool

**micsmc** a utility for monitoring the physical parameters of Intel Xeon Phi coprocessors: model, memory, core rail temperatures, core frequency, power usage, etc.

**micctrl** a comprehensive configuration tool for the Intel Xeon Phi coprocessor operating system

**miccheck** a set of diagnostic tests for the verification of the Intel Xeon Phi coprocessor configuration

**micrasd** a host daemon logger of hardware errors reported by Intel Xeon Phi coprocessors

**micflash** an Intel Xeon Phi flash memory agent
micsmc: Real-Time Monitoring Tool
§2. Programming Models for Intel Xeon Phi Applications

Native Applications and MPI
Hello (Parallel) World

Application can be compiled for host and for coprocessor:

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

int main(){
    printf("Hello world! I have %ld logical cores.\n", 
    sysconf(_SC_NPROCESSORS_ONLN ));
}
```

Compile and run on host:

```bash
user@host% icc hello.c
user@host% ./a.out
Hello world! I have 32 logical cores.
user@host%
```
Hello (Parallel) World

Compile and run the same code on the coprocessor in the native mode:

```
user@host% icc hello.c -mmic
user@host% scp a.out mic0:~/
a.out 100% 10KB 10.4KB/s 00:00
user@host% ssh mic0
user@mic0% pwd
/home/user
user@mic0% ls
a.out
user@mic0% ./a.out
Hello world! I have 240 logical cores.
user@mic0%
```

- Use `--mmic` to produce executable for MIC architecture
- Must transfer the executable to the coprocessor
- Can run on the coprocessor from the shell
Native applications for coprocessors with MPI

“Hello World” in MPI:

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[]) {
    int i, rank, size, namelen;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name (name, &namelen);
    printf ("Hello World from rank %d running on %s!\n", rank, name);
    if (rank == 0) printf("MPI World size = %d processes\n", size);
    MPI_Finalize ();
}
```
Running MPI applications on host

Set up MPI environment variables

- Use wrapper script `mpiicpc` to compile
- Use automated tool `mpirun` to launch

```
user@host% source /opt/intel/impi/4.1.0/intel64/bin/mpivars.sh
user@host% mpiicpc -o HelloMPI.XEON HelloMPI.c
user@host% mpirun -host localhost -np 2 ./HelloMPI.XEON
Hello World from rank 1 running on host!
Hello World from rank 0 running on host!
MPI World size = 2 processes
```
Running native MPI applications on coprocessor

```
user@host% source /opt/intel/impi/4.1.0/intel64/bin/mpivars.sh
user@host% export I_MPI_MIC=1
user@host% mpiicpc -mmic -o HelloMPI.MIC HelloMPI.c
user@host% scp HelloMPI.MIC mic0:~/
user@host% mpirun -host mic0 -np 2 ~/HelloMPI.MIC
```

Hello World from rank 1 running on host-mic0!
Hello World from rank 0 running on host-mic0!
MPI World size = 2 processes

- Set up MPI environment variables *and* `I_MPI_MIC=1`
- NFS-share the MPI library with coprocessor (not shown)
- Use wrapper script `mpiicpc` with argument `-mmic` to compile
- Copy or NFS-share the executable to the coprocessor file system
- Use automated tool `mpirun` to launch
Explicit Offload Model
Explicit offload: pragma-based approach

“Hello World” in the explicit offload model:

```c
#include <stdio.h>
int main(int argc, char * argv[]) {
    printf("Hello World from host!\n");
    #pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n"); fflush(0);
    }
    printf("Bye\n");
}
```
Compiling and running an offload application

```bash
user@host% icpc hello_offload.cpp -o hello_offload
user@host% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
```

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- Code inside of `#pragma offload` is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside `#pragma offload` runs on the host system
Offloading functions

```c
__attribute__((target(mic))) void MyFunction() {
    // ... implement function as usual
}

int main(int argc, char * argv[]) {
    #pragma offload target(mic)
    {
        MyFunction();
    }
}
```

- Functions used on coprocessor must be marked with the specifier `__attribute__((target(mic)))`
- Compiler produces a host version and a coprocessor version of such functions (to enable fall-back to host)
Offloading multiple functions

```c
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
    // ... implement function as usual
}

void MyFunctionTwo() {
    // ... implement function as usual
}
#pragma offload_attribute(pop)
```

- To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`
void MyFunction() {
    const int N = 1000;
    int data[N];
    #pragma offload target(mic)
    {
        for (int i = 0; i < N; i++)
            data[i] = 0;
    }
}

- To offload scope-local scalars and known-size arrays, nothing needs to be done
- Data is copied from host to coprocessor at the start of offload
- Data is copied back from coprocessor to host at the end of offload
- Bitwise-copyable data only (arrays of basic types and scalars)
  C++ classes, etc. should use virtual shared memory model (MYO)
Global and static variables must be marked with the offload attribute

#pragma offload_attribute(push/pop) may be used as well
Data Marshalling for Dynamically Allocated Data

```c
double *p1=(double*)malloc(sizeof(double)*N);
double *p2=(double*)malloc(sizeof(double)*N);

#pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
{
    // ... perform operations on p1[] and p2[]
}
```

- `#pragma offload` recognizes clauses `in`, `out`, `inout` and `nocopy`
- Data size (length), alignment, redirection, and other properties may be specified
- Marshalling is required for pointer-based data
- Bitwise-copyable data only (arrays of basic types and scalars)
  C++ classes, etc. should use virtual shared memory model (MYO)
Virtual-shared memory model

```c
_Cilk_shared int arr[N]; // This is a virtual-shared array

_Cilk_shared void Compute() { // This function may be offloaded
    // ... function uses array arr[]
}

int main() {
    // arr[] can be initialized on the host
    _Cilk_offload Compute(); // and used on coprocessor
    // and the values are returned to the host
}
```

- Two keywords: `_Cilk_shared` and `_Cilk_offload`
- Data is synced from host to coprocessor before the start of offload
- Data is synced from coprocessor to host at the end of offload
- Only modified data is transferred
Dynamic allocation of virtual-shared memory

```c
int* _Cilk_shared data;

int main() {
    data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));

    _Cilk_Offload SomeFunction(data);

    _Offload_shared_free(data);
}
```

- Mark pointers to virtual-shared data with _Cilk_shared
- Use _Offload_shared_malloc()/_Offload_shared_free()
Asynchronous offload

- By default, `#pragma offload` blocks until offload completes
- Use clause “signal” to avoid blocking
- Use `#pragma offload_wait` to block where needed

```c
char* offload0;

#pragma offload target(mic:0) signal(offload0) in(data : length(N))
{ /* ... will not block code execution because of clause "signal" */ }

DoSomethingElse();

/* Now block until offload signalled by pointer "offload0" completes */
#pragma offload_wait target(mic:0) wait(offload0)
```

- Any pointer can be used as a signal
- Use the target number to avoid hanging
Target-specific code

- Functions with the offload attribute are compiled twice: once for the host architecture, once for the MIC (coprocessor) architecture.
- During coprocessor compilation, preprocessor macro `__MIC__` is defined.
- Allows to fine-tune application performance or output where necessary.

```c
void __attribute__((target(mic))) MyFunction() {
    #ifdef __MIC__
        printf("I am running on a coprocessor.\n");
        const int tuningParameter = 16;
    #else
        printf("I am running on the host.\n");
        const int tuningParameter = 8;
    #endif
    // ... Proceed, using the variable tuningParameter
}
```
Multiple Coprocessors and Clusters
Clusters with Intel Xeon Phi coprocessors

- Option 1: run MPI processes on hosts and offload to coprocessors:

- Option 2: MPI as native applications on hosts and coprocessors:
Multiple coprocessors with explicit offload

- **Querying the number of coprocessors:**

```c
const int numDevices = _Offload_number_of_devices();
printf("Number of available coprocessors: %d\n", numDevices);
```

- **Specifying offload target:**

```c
#pragma offload target(mic: 0)
{ /* ... */ }
```
Multiple blocking offloads using host threads (explicit offload)

```c
const int nDevices = _Offload_number_of_devices();
#pragma omp parallel for
for (int i = 0; i < nDevices; i++) {
#pragma offload target(mic: i)
    {
        MyFunction(/*...*/);
    }
}
```

- Up to 8 coprocessors, up to 32 host threads
- All offloads start simultaneously and block the respective thread
### Blocking explicit offloads using threads: dynamic work distribution across coprocessors

```c
const int nDevices = _Offload_number_of_devices();
omp_set_num_threads(nDevices);
#pragma omp parallel for schedule(dynamic, 1)
for (int i = 0; i < nWorkItems; i++) {
    const int iDevice = omp_get_thread_num();
    #pragma offload target(mic: iDevice)
    {
        MyFunction(i);
    }
}
```

- Up to 8 coprocessors, up to 32 host threads
- `nWorkItems` are dynamically scheduled on `nDevices`
Multiple asynchronous explicit offloads from a single thread

```c
const int nDevices = _Offload_number_of_devices();
char sig[nDevices];
for (int i = 0; i < nDevices; i++) {
    #pragma offload target(mic: i) signal(&sig[i])
    {
        MyFunction(/*...*/);
    }
}
for (int i = 0; i < nDevices; i++) {
    #pragma offload_wait target(mic: i) wait(&sig[i])
}
```

- Any pointer acts as a signal
- Must wait for all signals
Heterogeneous MPI applications: host + coprocessors

```
user@host% mpirun -host mic0 -n 2 ~/Hello.MIC : -host mic1 -n 2 ~/Hello.MIC :
% -host localhost -n 2 ~/Hello.XEON
Hello World from rank 5 running on localhost!
Hello World from rank 4 running on localhost!
Hello World from rank 2 running on mic1!
Hello World from rank 3 running on mic1!
Hello World from rank 1 running on mic0!
Hello World from rank 0 running on mic0!
MPI World size = 6 ranks
```

- Specify Xeon executable for host processes
- Specify Xeon Phi executable for coprocessor processes
Section 2 Review:
Programming Models for Intel Xeon Phi applications
Programming models

1. Native coprocessor applications
   - Compile with -mmic
   - Run with micnativeloadex or scp+ssh
   - The way to go for MPI applications without offload

2. Explicit offload
   - Functions, global variables require __attribute__((target(mic)))
   - Initiate offload, data marshalling with #pragma offload
   - Only bitwise-copyable data can be shared

3. Clusters and multiple coprocessors
   - #pragma offload target(mic:i)
   - Use threads to offload to multiple coprocessors
   - Run native MPI applications