Introduction to Intel Xeon Phi Coprocessors – Part I

Summer Supercomputing Institute
June 19th 2014

Carlos Rosales
carlos@tacc.utexas.edu
Breakdown

• Introduction
  – Architecture description
  – Using the Xeon Phi

• Native Execution
  – Environment
  – Building and execution of native apps

• Distributed execution
  – MPI
  – MPI+OMP

• Offload
  – Compiler Assisted
  – Automatic
  – MPI+Offload
Xeon Phi — MIC

- Xeon Phi = first product of Intel’s Many Integrated Core (MIC) architecture
- Co-processor
  - PCI Express card
  - Stripped down Linux operating system (busybox)
- Dense, simplified processor
  - Many power-hungry operations removed
  - Wider vector unit
  - Higher hardware thread count
- Lots of names
  - Many Integrated Core architecture, aka MIC
  - Knights Corner, aka KNC (code name)
  - Intel Xeon Phi Co-processor SE10P (product name)
Leverage x86 architecture (CPU with many cores)
- x86 cores that are simpler, but allow for more compute throughput
Leverage existing x86 programming models
Dedicate much of the silicon to floating point ops
Cache coherent
Increase floating-point throughput
Strip expensive features
- out-of-order execution
- branch prediction
Widen SIMD registers for more throughput
Fast (GDDR5) memory on card
Intel Xeon Phi Chip

- 22 nm process
- Based on what Intel learned from
  - Larrabee
  - SCC
  - TeraFlops Research Chip
MIC Architecture

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

- PPF
- PF
- D0
- D1
- D2
- E
- WB
- D2
- E
- VC1
- VC2
- V1-V4
- WB

**D2**
- DEC
- VPU
- RF
- 3R, 1W

**E**
- LD
- EMU

**VC1**
- ST

**VC2**
- Mask
- RF
- Scatter
- Gather

**V1-V4**
- Vector ALUs
  - 16 Wide x 32 bit
  - 8 Wide x 64 bit
- Fused Multiply Add

George Chrysos, Intel, Hot Chips 24 (2012):
**Phi SE10P**
- **Processor**
  - 61 cores
  - ~1.1 GHz
  - 512-bit wide vector unit
  - 1.074 TF peak DP
- **Data Cache**
  - L1 32KB/core
  - L2 512KB/core, 30.5 MB/chip
- **Memory**
  - 8GB GDDR5 DRAM
  - 352 GB/s Peak
- **PCIe**
  - 5.0 GT/s, 16-bit

**Sandy Bridge E5-2680**
- **Processor**
  - 16 cores
  - ~2.7-3.5 GHz
  - 256-bit width vector unit
  - 224 GF peak DP
- **Data Cache**
  - L1 32KB/core
  - L2 256KB/core
  - L3 20 MB
- **Memory**
  - 32 GB DDR3
  - 51.2 GB/s per socket
Advantages

• 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
  – “Optimize once, run anywhere”
  – Our MIC porting efforts for scientific codes are frequently doubling performance on Sandy Bridge.
Phi Programming Models

• Native Phi
  – Use one Phi and run OpenMP, MPI, or MPI+X programs directly

• Symmetric Execution
  – MPI on Host and Phi
  – MPI+X almost mandatory due to memory restrictions

• Offload to Xeon Phi
  – Targeted offload through OpenMP extensions
  – Automatically offload some MKL routines
  – Use MPI on host + offload for distributed execution
Native Execution

• Build for Phi with –mmic

• Execute on host (runtime will automatically detect a Phi executable)

• … or ssh to mic0 and run on the Phi

• Can safely use all 61 cores
  – Offload programs should stay away from the 61st core since the offload daemon runs here
Symmetric MPI

- Host and Phi can operate symmetrically as MPI targets
  - High code reuse
  - MPI and hybrid MPI+X (X = OpenMP, Cilk+, TBB, pthreads)

- Need to balance workload between big and little cores

- Careful to create locality between local host, local Phi, remote hosts, and remote Phis

- Take advantage of topology-aware MPI interface under development in MVAPICH2
  - NSF STCI project with OSU, TACC, and SDSC
Offload to Phi

- Existing codes using accelerators have already identified regions where offload works well

- Porting these to OpenMP offload is typically straightforward

- Automatic offload where MKL kernel routines can be used
  - xGEMM, etc.
Will My Code Run on Xeon Phi?

• Yes
• … but that’s the wrong question
  – Will your code run *best* on Phi?, or
  – Will you get great Phi performance without additional work? (The answer is most likely NO)
Phi Programming Experiences at TACC

• Codes port easily
  – Minutes to days depending mostly on library dependencies

• Performance can require real work
  – While the software environment continues to evolve
  – Getting codes to run *at all* is almost too easy; really need to put in the effort to get what you expect

• Scalability is pretty good
  – Multiple threads per core is really important
  – Getting your code to vectorize is really important
LBM Native

- Lattice Boltzmann Method CFD code
  - Carlos Rosales, TACC
  - MPI+OMP code

- Finding all the right routines to parallelize is critical

- Achieves > 2x compared to 2 SNB sockets
PETSc/MUMPS with AO

- Hydrostatic ice sheet modeling
- MUMPS solver (called through PETSC)
- BLAS calls automatically offloaded behind the scenes

3D Geometric Multigrid code (PETSc/MUMPS)
NATIVE EXECUTION
Overview

- Why run native?
- What is a native application?
- Building a native application
- Running a native application

- Setting affinity and pinning tasks
- Optimization
  - Vectorization
  - Alignment
  - Parallelization
What is a native application?

- It is an application built to run exclusively on the MIC coprocessor.

- MIC is not binary compatible with the host processor
  - Instruction set is *similar* to Pentium, but not all 64 bit scalar extensions are included.
  - MIC has 512 bit vector extensions, but does NOT have MMX, SSE, or AVX extensions.

- Native applications can’t be used on the host CPU, and vice versa.
Why run a native application?

- It is possible to login and run applications on the MIC without any host intervention

- Easy way to get acquainted with the properties of the MIC
  - Performance studies
  - Single card scaling tests (OMP/MPI)
  - No issues with data exchange with host

- The native code probably performs quite well on the host CPU once you build a host version
  - Good path for symmetric runs (next talk)
Building a native application

- Cross-compile on the host (login or compute nodes)
  - No compilers installed on coprocessors
- MIC is fully supported by the Intel C/C++ and Fortran compilers (v13+):

  ```
  icc -openmp -mmic mysource.c -o myapp.mic
  ifort -openmp -mmic mysource.f90 -o myapp.mic
  ```

- The `-mmic` flag causes the compiler to generate a native mic executable
- It is convenient to use a `.mic` extension to differentiate MIC executables
Running a native application

• Options to run from mic0 from a compute node:
  1. Traditional ssh remote command execution
     • `c422-703% ssh mic0 ls`
     • Clumsy if environment variables or directory changes needed
  2. Interactively login to mic:
     • `c422-703% ssh mic0`
     • Then use as a normal server
  3. Explicit launcher:
     • `c422-703% micrun ./a.out.mic`
  4. Implicit launcher:
     • `c422-703% ./a.out.mic`
The micrun launcher has three nice features:

- It propagates the current working directory
- It propagates the shell environment (with translation)
  - Environment variables that need to be different on host and coprocessor need to be defined using the MIC_ prefix on the host. E.g.,
    - `c422-703% export MIC_OMP_NUMTHREADS=183`
    - `c422-703% export MIC_KMP_AFFINITY="verbose,balanced"`
- It propagates the command return code back to the host shell

These features work whether the launcher is used explicitly or implicitly
Environmental Variables on the MIC

• If you ssh to mic0 and run directly from the card use the regular names:
  – OMP_NUM_THREADS
  – KMP_AFFINITY
  – I_MPI_PIN_PROCESSOR_LIST
  – ...

• If you use the launcher, use the MIC_ prefix to define them on the host:
  – MIC_OMP_NUM_THREADS
  – MIC_KMP_AFFINITY
  – MIC_I_MPI_PIN_PROCESSOR_LIST
  – ...

• You can also define a different prefix:
  – export MIC_ENV_PREFIX=MYMIC
  – MYMIC_OMP_NUM_THREADS
  – ...

Native Execution Quirks

- The mic runs a lightweight version of Linux, based on BusyBox
  - Some tools are missing: `w`, `numactl`
  - Some tools have reduced functionality: `ps`

- Relatively few libraries have been ported to the coprocessor environment

- These issues make the implicit or explicit launcher approach even more convenient
Best Practices For Running Native Apps

• Always bind processes to cores
  – For MPI tasks (more on next presentation)
    • `I_MPI_PIN`
    • `I_MPI_PIN_MODE`
    • `I_MPI_PIN_PROCESSOR_LIST`
  – For threads
    • `KMP_AFFINITY={compact, scatter, balanced}`
    • `KMP_AFFINITY=explicit,proclist=[0,1,2,3,4]`
    • Adding `verbose` will dump the full affinity information when the run starts
    • Adding `granularity=fine` binds to specific thread contexts and may help in codes with heavy L1 cache reuse

• The MIC is a single chip, so there is no need for `numactl`
• If other affinity options can’t be used the command `taskset` is available.
KMP_AFFINITY Example

compact

balanced

scatter
Logical to Physical Processor Mapping

- **Hardware:**
  - Physical Cores are 0..60
  - Logical Cores are 0..243

- **Mapping is not what you are used to!**
  - Logical Core 0 maps to Physical core 60, thread context 0
  - Logical Core 1 maps to Physical core 0, thread context 0
  - Logical Core 2 maps to Physical core 0, thread context 1
  - Logical Core 3 maps to Physical core 0, thread context 2
  - Logical Core 4 maps to Physical core 0, thread context 3
  - Logical Core 5 maps to Physical core 1, thread context 0
  - Logical Core 240 maps to Physical core 59, thread context 3
  - Logical Core 241 maps to Physical core 60, thread context 1
  - Logical Core 242 maps to Physical core 60, thread context 2
  - Logical Core 243 maps to Physical core 60, thread context 3

- **OpenMP threads start binding to logical core 1, not logical core 0**
  - For compact mapping 240 OpenMP threads are mapped to the first 60 cores
    - No contention for the core containing logical core 0 – the core that the O/S uses most
  - But for scatter and balanced mappings, contention for logical core 0 begins at 61 threads
    - Not much performance impact unless O/S is very busy
    - Best to avoid core 60 for offload jobs & MPI jobs with compute/communication overlap
How Do I Tune Native Applications?

• Vectorization and Parallelization are critical!
  – Single-thread scalar performance: ~1 GHz Pentium

• Vector width is 512 bits
  – 8 double precision values / 16 single precision values
  – You don’t want to lose factors of 8-16 in performance

• Compiler reports provide important information about effectiveness of compiler at vectorization
  – Start with a simple code – the compiler reports can be very long & hard to follow
  – There are lots of options & reports! Details at:
Vectorization Compiler reports

• Option `-vec-report3` gives diagnostic information about every loop, including
  – Loops successfully vectorized (also at `-vec-report1`)
  – Loops not vectorized & reasons (also at `-vec-report2`)
  – Specific dependency info for failures to vectorize
  – Option `-vec-report6` provides additional info:
    • Array alignment for each loop
    • Unrolling depth for each loop

• Quirks
  – Functions typically have most/all of the vectorization messages repeated with the line number of the call site – ignore these and look at the messages with the line number of the actual loop
  – Reported reasons for not vectorizing are not very helpful – look at specific dependency info & remember about C aliasing rules
vec-report  Example

• Code: STREAM Copy kernel
  
  ```c
  #pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE; j++)
    c[j] = a[j];
  ```

• `vec-report` messages
  
  ```
  stream_5-10.c(354): (col. 6) remark: vectorization support: reference c has aligned access.
  stream_5-10.c(354): (col. 6) remark: vectorization support: reference a has aligned access.
  stream_5-10.c(354): (col. 6) remark: vectorization support: streaming store was generated for c.
  stream_5-10.c(353): (col. 2) remark: LOOP WAS VECTORIZED.
  stream_5-10.c(354): (col. 6) remark: vectorization support: reference c has unaligned access.
  stream_5-10.c(354): (col. 6) remark: vectorization support: reference a has unaligned access.
  stream_5-10.c(354): (col. 6) remark: vectorization support: unaligned access used inside loop body.
  stream_5-10.c(353): (col. 2) remark: loop was not vectorized: vectorization possible but seems inefficient.
  ```

• Many other combinations of messages are possible
  
  – Remember that OpenMP will split loops in ways that can break 64-Byte alignment – alignment depends on thread count
- Option `-opt-report-phase hpo` provides good info on OpenMP parallelization.

- Option `-opt-report-phase hlo` provides info on software prefetching.

- Option `-opt-report 1` gives a medium level of detail from all compiler phases, split up by routine.

- Option `-opt-report-file=filename` saves the lengthy optimization report output to a file.
Tuning Limitations

- Currently there is no support for `gprof` when compiling native applications.

- Profiling is supported by Intel’s `Vtune` product:
  - But this is not currently enabled on Stampede.
  - Vtune is a complex profiling software that deserves its own training session.
Performance Tuning Notes (1)

- Xeon Phi has multiple threads per core (NOT HT)
  - Four thread contexts per physical core
  - Registers are replicated
  - L1D, L1I, and (private, unified) L2 caches are shared

- Vector Unit instruction issue limitation:
  - The vector unit can issue one instruction per cycle
  - L1D Cache can deliver 64 Bytes (1 vector register) every cycle
  - But a thread can issue a vector instruction every other cycle
  - Need at least two threads to fully utilize the vector unit
  - Using 3-4 threads does not increase maximum issue rate, but often helps tolerate latency
George Chrysos, Intel, Hot Chips 24 (2012):
Vector Processing Unit

George Chrysos, Intel, Hot Chips 24 (2012):


Vector ALUs
16 Wide x 32 bit
8 Wide x 64 bit
Fused Multiply Add
Performance Tuning Notes (2)

- Cache Hierarchy:
  - L1I and L1D are 32kB, 8-way associative, 64-Byte cache lines
    - Same sizes & associativity as Xeon E5 (“Sandy Bridge”), but shared when using multiple threads/core
    - 1 cycle latency for scalar loads, 3 cycles for vector loads
  - L2 (unified, private) is 512kB, 8-way associative, 64-Byte lines
    - Latency ~25 cycles (idle), increases under load
    - Bandwidth is 1 cache line every other cycle
  - All 60 of the other L2 caches are snooped on an L2 cache miss
    - Clean or Dirty data will be transferred to requestor’s L1D
    - This eliminates load from DRAM on shared data accesses
    - Cache-to-Cache transfers are about 275ns, independent of relative core numbers
Performance Tuning Notes (3)

- Idle Memory Latency is ~275-280 ns

- Required Concurrency:
  - $277 \text{ ns} \times 352 \text{ GB/s} = 97,504 \text{ Bytes} = 1524 \text{ cache lines}$
    - This is ~25 concurrent cache misses per core
    - Theoretically supported by the HW, but not attainable in practice
    - The actual number increases under load as the latency increases

- Hardware Prefetch
  - No L1 prefetchers
  - Simplified L2 prefetcher
    - Only identifies strides up to 2 cache lines
    - Prefetches up to 4 cache-line-pairs per stream
    - Monitors up to 16 streams (on different 4kB pages)
      - These are shared by the hardware threads on a core

- Software prefetch is often required to obtain good bandwidth
Software Prefetch vs Data Location

• Xeon Phi can only issue one vector instruction every other cycle from a single thread context, so:
  – If data is already in the L1 Cache, Vector Prefetch instructions use up valuable instruction issue bandwidth
  – But, if data is in the L2 cache or memory, Vector Prefetch instructions provide significant increases in sustained performance.

• The next slide shows the effect of including vector prefetch instructions (default with “-O3”) vs excluding them (with “-no-opt-prefetch”)
  – Data is L1 contained for array sizes of 2k elements or less
  – Data is L2-contained for array sizes of ~32k elements or less
Effect of SW Prefetch with Data on Cache

Stream2 DAXPY on Xeon Phi SE10P: Effect of Software Prefetch on Performance with Data in Cache
Intel reference material

- Main Software Developers web page:
- A list of links to very good training material at:
- Many answers can also be found in the Intel forums:
- Specific information about building and running “native” applications:
- Debugging:
More Intel reference Material

• Search for these at [www.intel.com](http://www.intel.com) by document number
  – This is more likely to get the most recent version than searching for the document number via Google.

• Primary Reference:
  – “Intel Xeon Phi Coprocessor System Software Developers Guide” (document 488596 or 328207)

• Advanced Topics:
  – “Intel Xeon Phi Coprocessor (codename: Knights Corner) Performance Monitoring Units” (document 327357)

• WARNING:
  – Intel sometimes describes the number of vector registers as 16 in this documents. The actual number is 32.
Questions?

For more information:

www.tacc.utexas.edu