Introduction to Intel Xeon Phi Coprocessors

Slides by Johannes Henning
Xeon Phi Coprocessor Lineup

7 Family
Highest Performance
Most Memory
Performance leadership

- 16GB GDDR5
- 352GB/s
- >1.2TF DP
- 300W TDP

5 Family
Optimized for High Density Environments
Performance/Watt leadership

- 8GB GDDR5
- >300GB/s
- >1TF DP
- 225-245W TDP

3 Family
Outstanding Parallel Computing Solution
Performance/$ leadership

- 6GB GDDR5
- 240GB/s
- >1TF DP
- 300W TDP

Optional 3-year Warranty

Extend to 3-year warranty on any Intel® Xeon Phi™ Coprocessor.
Product Code: XPX100WRNTY, MM# 933057

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/
Xeon Phi 5110P Specifications

60 Cores based on P54C architecture (Pentium)
- > 1.0 Ghz clock speed; 64bit based x86 instructions + SIMD
- **No x86 compatibility**, uses own MIC instruction set
- 1x 25 MB L2 Cache (=512KB per core) + 64 KB L1
  - Cache coherency
- 8 GB of GDDR5 Memory
- **In-Order Execution**
- **4 Hardware Threads per Core** (240 logical cores)
  - Think graphics-card hardware threads
  - Only one runs = memory latency hiding
  - Switched after each instruction
- 512 bit wide VPU with new ISA KCi
  - **No support for MMX, SSE or AVX**
  - Could handle 8 double precision floats/16 single precision floats
  - Always structured in vectors with 16 elements
### Xeon Phi Performance in Practice: Case Studies

<table>
<thead>
<tr>
<th>Segment</th>
<th>Application/Code</th>
<th>Performance vs. 2S Xeon*</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCC</td>
<td>NEC / Video Transcoding (see case study: NEC Case Study)</td>
<td>Up to 3.0x²</td>
</tr>
<tr>
<td>Energy</td>
<td>Seismic Imaging ISO3DFD Proxy 16th order Isotropic kernel RTM</td>
<td>Up to 1.45x³</td>
</tr>
<tr>
<td></td>
<td>Seismic Imaging 3DFD TTI 3- Proxy 8th order RTM (complex structures)</td>
<td>Up to 1.23x³</td>
</tr>
<tr>
<td></td>
<td>Petrobras Seismic ISO-3D RTM (with 1, 2, 3 or 4 Intel® Xeon Phi™ coprocessors)</td>
<td>Up to 2.2x, 3.4x, 4.6x or 5.6x⁴</td>
</tr>
<tr>
<td>Financial Services</td>
<td>BlackScholes SP / DP  &lt;br&gt; Monte Carlo European Option SP / DP  &lt;br&gt; Monte Carlo RNG European SP / DP  &lt;br&gt; Binomial Options SP / DP</td>
<td>SP: Up to 2.12x³; DP Up to 1.72x³&lt;br&gt;SP: Up to 7x³; DP Up to 3.13x³&lt;br&gt;SP: Up to 1.58x³; DP Up to 1.17x³&lt;br&gt;SP: Up to 1.85x³; DP Up to 1.85x³</td>
</tr>
<tr>
<td>Life Science</td>
<td>BWA/Bio-Informatics  &lt;br&gt; Wayne State University/MPI-Hmmer  &lt;br&gt; GROMACS /Molecular Dynamics</td>
<td>Up to 1.5x⁴  &lt;br&gt; Up to 1.56x¹  &lt;br&gt; Up to 1.36x¹</td>
</tr>
<tr>
<td>Manufacturing</td>
<td>ANSYS / Mechanical SMP  &lt;br&gt; Sandia Mantevo / miniFE case study: software.intel.com/en-us/articles/running-minife-on-intel-xeon-phi-coprocessors</td>
<td>Up to 1.88x⁶  &lt;br&gt; Up to 2.3x⁴</td>
</tr>
<tr>
<td>Physics</td>
<td>ZIB (Zuse-Institut Berlin) / Ising 3D (Solid State Physics)  &lt;br&gt; ASKAP tHogbomClean (astronomy)  &lt;br&gt; Princeton / GTC-P (Gyrokinetic Torodial) Turbulence Simulation IVB</td>
<td>Up to 3.46x¹  &lt;br&gt; Up to 1.73x³  &lt;br&gt; Up to 1.18x⁶</td>
</tr>
<tr>
<td>Weather</td>
<td>WRF /Code WRF V3.5</td>
<td>1.56x⁶</td>
</tr>
</tbody>
</table>

1. 25 Xeon E5 2670 vs. 25 Xeon® E5 2670 + 1 Xeon Phi™ coprocessor (Symmetric)
2. 25 Xeon E5 2670 vs. 25 Xeon E5 2670 +2 Xeon Phi™ coprocessor
3. 25 Xeon E5-2697v2 vs. 1 Xeon Phi™ coprocessor (Native Mode)
4. 25 Xeon E5-2697v2 vs. 25 Xeon E5 2697v2 +1 Xeon Phi™ coprocessor (Symmetric Mode) (for Petrobras, 1, 2, 3 or 4 Xeon Phi’s in the system)
5. 25 Xeon E5 2670 vs. 25 Xeon® E5 2670 + 1 Xeon Phi™ coprocessor (Symmetric) (only 2 Xeon cores used to optimize licensing costs)
6. 4 nodes of 25 E5-2697v2 vs. 4 nodes of E5-2697v2 + 1 Xeon Phi™ coprocessor (Symmetric)

* Xeon = Intel® Xeon® processor  
* Xeon Phi = Intel® Xeon Phi™ coprocessor

Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/
Operating System

- Minimal, embedded Linux
- Linux Standard Base (LSB) Core libraries.
- Implements Busybox minimal shell environment
Operating System

```
tesla:/ # ssh mic0
[root@tesla-mic0 /root]# cat /proc/cpuinfo | tail -n 22
model          : 1
model name     : 0b/01
stepping       : 3
cpu MHz        : 1052.630
cache size     : 512 KB
physical id    : 0
siblings       : 240
core id        : 59
cpu cores      : 60
apicid         : 239
initial apicid : 239
fpu            : yes
fpu exception  : yes
cpuuid level   : 4
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic mtrr mca pat fxt8 ht syscall nx lm re p_good nopl lahf_lm
bogomips       : 2114.13
clflush size   : 64
cache_alignment: 64
address sizes  : 40 bits physical, 48 bits virtual
power management:
[root@tesla-mic0 /root]#  
```
Hello World

```c
#include <stdio.h>

int main(void) {
    printf("Hello, World\n");
    return 0;
}
```

```bash
johannes.henning@tesla:~

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Offload-Compiler

- OpenMP like approach to "offload" functions onto the MIC
  - Like OpenMP pragma based library approach

```
int p1[SIZE] = populate();
int p2[SIZE] = populate();
int res[SIZE];
#pragma offload target(mic) in(p1, p2:length(SIZE))
out(res)
{
  for (i=0; i<SIZE; i++){
    res[i] = p1[i] + p2[i];
  }
}
```

- Simple mechanism but questionable optimisation
- Very limited in features
Other Keywords:

- `__attribute__((target(mic)))` int res[];
- `#pragma offload_attribute(push, target(mic))`
- `#pragma offload_attribute(pop)`
- nocopy()
- alloc_if()
- free_if()
- `#pragma offload_transfer`
- `#pragma offload_wait`
- signal() wait()

Offloading mode:

```c
#pragma omp target (mic)
#pragma omp parallel for reduction(+:pi)
for (i=0; i<count; i++) {
    float t = (float)((i+0.5f)/count);
    pi += 4.0f/(1.0f+t*t);
}
pi /= count;
```

You can also use OpenMP in native Phi-binaries

OpenMP 4.1 will integrate offload clauses into omp pragmas
Intel TBB

- Template based C++ library
- Task parallelism + Task stealing
- API Overview:
  - parallel_for, parallel_reduce, parallel_scan
  - parallel_while, parallel_do, parallel_pipeline, parallel_sort
  - concurrent_queue, ...
  - mutex, spin_mutex, ...
  - fetch_and_add, fetch_and_increment, ...
void SerialApplyFoo(float a[], size_t n) {
    for (size_t i=0; i!=n; ++i)
        Foo(a[i]);
}
#include "tbb/tbb.h"

using namespace tbb;

class ApplyFoo {
    float *const my_a;

public: void operator()( const blocked_range<size_t>& r ) const {
    float *a = my_a;
    for (size_t i=r.begin(); i!=r.end(); ++i)
        Foo(a[i]);
}

    ApplyFoo(float a[]) : my_a(a) {}
Intel Cilk Plus

- C++ extension providing:
  - fork-join task parallelism with work-stealing
  - auto vectorization
  - data-parallel language constructs for vectorization (array notation)
- Can/Should be combined with TBB or OpenMP
- Ease of usage bought with reduced fine tuning capabilities
Keywords
- cilk_spawn
- cilk_sync
- cilk_for

```c
#include <cilk/cilk.h>
void parallel_qsort(int * begin, int * end) {
  if (begin != end) {
    --end; // Exclude last element (pivot)
    int * middle = std::partition(begin, end, 
      std::bind2nd(std::less<int>(),*end));
    std::swap(*end, *middle); // pivot to middle
    cilk_spawn parallel_qsort(begin, middle);
    parallel_qsort(++middle, ++end); // Exclude pivot
  }
}
```
Reducers

```cpp
cilk::reducer_opadd<unsigned long long int> total (0);
cilk_for(unsigned int i = 1; i <= n; ++i) {
    *total += compute(i);
}
```

Holders

```cpp
cilk::holder<hash_table<K, V> > m_holder;
```
Intel Cilk Plus

- **Extension for array notation**

```plaintext
// refers to 12 elements in the two-dimensional array a
a[0:3][0:4]

// refers to elements 0 and 3 of the one-dimensional array b
b[0:2:3]

b[::] // refers to the entire array b

a[::] * b[::] // element-wise multiplication


a[0:4][1:2] + b[1:2][0:4] // error, different rank sizes

a[0:4][1:2] + b[0][1] // ok, adds a scalar b[0][1]

// to an array section.
```
OpenCL

- **Whats different to GPUs?**
  - Thread granularity bigger + No need for local shared memory

- **Work Groups are mapped to Threads**
  - 240 OpenCL hardware threads handle workgroups
  - More than 1000 Work Groups recommended
  - Each thread executes one work group

- **Implicit Vectorization by the compiler of the inner most loop**
  - 16 elements per Vector → dimension zero must be divisible by 16 otherwise scalar execution (Good WG-Size = 16)

```c
__Kernel ABC()
for (int i = 0; i < get_local_size(2); i++)
  for (int j = 0; j < get_local_size(1); j++)
    for (int k = 0; k < get_local_size(0); k++)
      Kernel_Body;
```

dimension zero of the NDRange
OpenCL

- Non uniform branching has significant overhead
  - Between workgroups it is okay, since each WG is executed by one thread
- Non Vector size alligned memory access has significant overhead
- Non linear access patterns have significant overhead

- Manual prefetching can be advantageous
- No hardware support for barriers

- However: Intel prefers
Each MIC is considered as a stand alone node

MICs can communicate directly via Infiniband (Hardware or Software)

Build and launch process:

```bash
$ source /opt/intel/composer_xe_2013.3.163/bin/compilervars.sh intel64
$ source /opt/intel/impi/4.1.1.036/bin64/mpivars.sh
$ mpiicc -mmic mpi_test.c -o mpi_test.mic
$ mpiicc mpi_test.c -o mpi_test.host
$ export I_MPI_MIC=enable
$ scp mpi_test.mic mic0:~/.
$ sudo scp /opt/intel/impi/4.1.1.036/mic/bin/* mic0:/bin/
$ sudo scp /opt/intel/impi/4.1.1.036/mic/lib/* mic0:/lib64/
$ sudo scp /opt/intel/composer_xe_2013.3.163/compiler/lib/mic/* mic0:/lib64/
$ mpirun -n <# of processes> -host <hostname1> <application> : -n <# of processes> -host <hostname2> <application>
```
Vectorization on Intel® compilers

- **Auto Vectorization**
  - Compiler knobs

- **Guided Vectorization**
  - Compiler hints/pragmas
  - Array notation

- **Low level Vectorization**
  - C/C++ vector classes
  - Intrinsics/Assembly

Auto vectorization: not all loops will vectorize

Data dependencies between iterations
- Proven Read-after-Write data (i.e., loop carried) dependencies
- Assumed data dependencies
  - Aggressive optimizations (e.g., IPO) might help

Vectorization won’t be efficient
- Compiler estimates how better the vectorized version will be
- Affected by data alignment, data layout, etc.

Unsupported loop structure
- While-loop, for-loop with unknown number of iterations
- Complex loops, unsupported data types, etc.
- (Some) function calls within loop bodies
  - Not the case for SVML functions

Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/

```c
for (int i = 0; i < N; i++)
    a[i] = a[i-1] + b[i];
```

```c
for (int i = 0; i < N; i++)
    a[c[i]] = b[d[i]];
```

```c
for (int i = 0; i < N; i++)
    a[i] = foo(b[i]);
```
Guided vectorization: disambiguation hints
Get rid of assumed vector dependencies

Assume function arguments won’t be aliased

- C/C++: Compile with `-fargument-noalias`

C99 “restrict” keyword for pointers
- Compile with `-restrict` otherwise

Ignore assumed vector dependencies (compiler directive)

- C/C++: `#pragma ivdep`
- Fortran: `!dir$ ivdep`

```c
void v_add(float *restrict c, float *restrict a, float *restrict b)
{
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

```c
void v_add(float *c, float *a, float *b)
{
    #pragma ivdep
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```
Guided vectorization: \#pragma simd

Force loop vectorization ignoring \texttt{all} dependencies

- Additional \texttt{clauses} for specify reductions, etc.

\begin{Verbatim}
void v_add(float *c, float *a, float *b)
{
    #pragma simd
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
\end{Verbatim}

\begin{Verbatim}
__declspec(vector)
void v_add(float c, float a, float b)
{
    c = a + b;
}
...
for (int i = 0; i < N; i++)
    v_add(C[i], A[i], B[i]);
\end{Verbatim}

Also supported in OpenMP

- Almost same functionality/syntax
  - Use \texttt{#pragma omp simd [clauses]} for SIMD loops
  - Use \texttt{#pragma omp declare simd [clauses]} for SIMD functions

- See \texttt{OpenMP 4.0 specification} for more information

Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/

SoSe 2015
#pragma ivdep versus #pragma simd

- **#pragma ivdep**
  - Implicit vectorization
  - Notifies the compiler about the absence of pointer aliasing
  - Based on practicability and costs, the compiler decides about vectorization

- **#pragma simd**
  - Explicit
  - Enforces vectorization regardless of the costs
  - If no parameter is provided, the vector length of the SIMD unit is assumed
Improving vectorization: data layout

Vectorization more efficient with unit strides
• Non-unit strides will generate gather/scatter
• Unit strides also better for data locality
• Compiler might refuse to vectorize

AoS vs SoA
• Layout your data as Structure of Arrays (SoA)

Traverse matrices in the right direction
• C/C++: a[i][:], Fortran: a(:,i)
• Loop interchange might help
  • Usually the compiler is smart enough to apply it
  • Check compiler optimization report

Array of Structures vs Structure of Arrays

```c
// Array of Structures (AoS)
struct coordinate {
    float x, y, z;
} crd[N];
...
for (int i = 0; i < N; i++)
    ... = ... f(crd[i].x, crd[i].y, crd[i].z);
```

Consecutive elements in memory
```
| x0 | y0 | z0 | x1 | y1 | z1 | ... | x(n-1) | y(n-1) | z(n-1) |
```

```c
// Structure of Arrays (SoA)
struct coordinate {
    float x[N], y[N], z[N];
} crd;
...
for (int i = 0; i < N; i++)
    ... = ... f(crd.x[i], crd.y[i], crd.z[i]);
```

Consecutive elements in memory
```
| x0 | x1 | ... | x(n-1) | y0 | y1 | ... | y(n-1) | z0 | z1 | ... | z(n-1) |
```
Improving vectorization: data alignment

Unaligned accesses might cause significant performance degradation

- Two instructions on current Intel® Xeon Phi™ coprocessor
- Might cause “false sharing” problems
  - Consumer/producer thread on the same cache line

Alignment is generally unknown at compile time

- Every vector access is potentially an unaligned access
  - Vector access size = cache line size (64-byte)
- Compiler might “peel” a few loop iterations
  - In general, only one array can be aligned, though

When possible, we have to

- Align our data
- Tell the compiler data is aligned
  - Might not be always the case

Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/
Does the Xeon Phi fit the problem?

Source: https://software.intel.com/en-us/articles/is-the-intel-xeon-phi-coprocessor-right-for-me
What you need to start

- An installation of the Intel Parallel Studio XE (>= 2013)

- Samples including build files at: /opt/intel/composerxe/Samples/

- The following lines in your bashrc
  
  ```bash
  if [ -d "/opt/intel/composer_xe_13/compiler/lib/intel64" ] ; then
  LD_LIBRARY_PATH="/opt/intel/composer_xe_13/compiler/lib/intel64:$LD_LIBRARY_PATH"
  fi
  
  if [ -d "/opt/intel/bin" ] ; then
  PATH="/opt/intel/bin:$PATH"
  fi
  ```

- MIC library files at: /opt/intel/composer_xe_\{version_number\}/compiler/lib/mic/
Knights Landing: Next Generation Xeon Phi

Architectural Enhancements = ManyX Performance

- Based on Intel® Atom™ core (based on Silvermont microarchitecture) with Enhancements for HPC
- 60+ cores
- 3+ Teraflops
- 3x Single-Thread
- 2-D Core Mesh
- High-Performance Memory
  - Over 5x STREAM vs. DDR4
  - Up to 16 GB at launch
- NUMA support

Server Processor

Core

- 14nm process technology
- 4 Threads/Core
- Deep Out-of-Order Buffers
- Gather/Scatter
- Better Branch Prediction
- Higher Cache Bandwidth
  - ... and many more

Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/
## Knights Landing – Some Details

### PERFORMANCE
- 3+ TeraFLOPS of double-precision peak theoretical performance per single socket node

### INTEGRATION
- **Intel® Omni Scale™ fabric integration**
  - Over 5x STREAM vs. DDR4\(^1\) → Over 400 GB/s
  - Up to 16GB at launch
  - NUMA support
  - Over 5x Energy Efficiency vs. GDDR5\(^2\)
  - Over 3x Density vs. GDDR5\(^2\)
  - In partnership with Micron Technology
  - Flexible memory modes including cache and flat

### SERVER PROCESSOR
- Standalone bootable processor (running host OS) and a PCIe coprocessor (PCIe end-point device)
- Platform memory: up to 384GB DDR4 using 6 channels
- Reliability (“Intel server-class reliability”)
- Power Efficiency (Over 25% better than discrete coprocessor)\(^4\) → Over 10 GF/W
- Density (3+ KNL with fabric in 1U)\(^5\)
- Up to 36 lanes PCIe* Gen 3.0

### MICROARCHITECTURE
- Over 8 billion transistors per die based on Intel’s 14 nanometer manufacturing technology
- Binary compatible with Intel® Xeon® Processors with support for Intel® Advanced Vector Extensions 512 (Intel® AVX-512)\(^6\)
- 3x Single-Thread Performance compared to Knights Corner\(^7\)
- 60+ cores in a 2D Mesh architecture
- 2 cores per tile with 2 vector processing units (VPU) per core
- 1MB L2 cache shared between 2 cores in a tile (cache-coherent)
- “Based on Intel® Atom™ core (based on Silvermont microarchitecture) with many HPC enhancements”
- 4 Threads / Core
- 2X Out-of-Order Buffer Depth\(^8\)
- Gather/scatter in hardware
- Advanced Branch Prediction
- High cache bandwidth
- 32KB Icache, Dcache
- 2 x 64B Load ports in Dcache
- 46/48 Physical/virtual address bits
- Most of today’s parallel optimizations carry forward to KNL
- Multiple NUMA domain support per socket

### AVAILABILITY
- First commercial HPC systems in 2H’15
- Knights Corner to Knights Landing upgrade program available today
- Intel Adams Pass board (1U half-width) is custom designed for Knights Landing (KNL) and will be available to system integrators for KNL launch; the board is OCP Open Rack 1.0 compliant, features 6 ch native DDR4 (1866/2133/2400MHz) and 36 lanes of integrated PCIe* Gen 3 I/O

---

\(^0\) Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/
AVX-512 – KNL and future XEON

- KNL and future Xeon architecture share a large set of instructions
  - but sets are not identical
- Subsets are represented by individual feature flags (CPUID)

Source: http://www.inteldevconference.com/lrz-hpc-code-modernization-workshop/
Large impact: Intel® AVX-512 instruction set

- Slightly different from future Intel® Xeon™ architecture AVX-512 extensions
- Includes SSE, AVX, AVX-2
- Apps built for HSW and earlier can run on KNL (few exceptions like TSX)
- Incompatible with 1st Generation Intel® Xeon™ Phi (KNC)

Medium impact: New, on-chip high bandwidth memory (MCDRAM) creates heterogeneous (NUMA) memory access

- can be used transparently too however

Minor impact: Differences in floating point execution / rounding due to FMA and new HW-accelerated transcendental functions - like exp()
Sources

- **Intel® Xeon Phi™ System Software Developer’s Guide**

- Intel® Xeon Phi™ Coprocessor DEVELOPER’S QUICK START GUIDE

- Intel® Many Integrated Core Platform Software Stack

- OpenCL Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor

- Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual

- An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors

- Using the Intel® MPI Library on Intel® Xeon Phi™ Coprocessor Systems

- Understanding MPI Load Imbalance with Intel ® Trace Analyzer and Collector

- A Guide to Vectorization with Intel® C++ Compilers

- Differences in floating-point arithmetic between Intel® Xeon® processors and the Intel® Xeon® Phi™

- Intel ® Xeon Phi™ Coprocessor Datasheet

- Intel® Many Integrated Core Symmetric Communications Interface (SCIF) User Guide

- Intel ® HPC Code Modernization Workshop